Mario Bravetti
Tevfik Bultan (Eds.)

LNCS 6551

Web Services and Formal Methods

7th International Workshop, WS-FM 2010
Hoboken, NJ, USA, September 2010
Revised Selected Papers

Lecture Notes in Computer Science 6551

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

Mario Bravetti Tevfik Bultan (Eds.)

Web Services and Formal Methods

7th International Workshop, WS-FM 2010
Hoboken, NJ, USA, September 16-17, 2010
Revised Selected Papers

Springer

Volume Editors

Mario Bravetti
Dipartimento di Scienze dell' Informazione
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: bravetti@cs.unibo.it

Tevfik Bultan
University of California, Department of Computer Science
Santa Barbara, CA 93106-5110, USA
E-mail: bultan@cs.ucsb.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19588-4 e-ISBN 978-3-642-19589-1
DOI 10.1007/978-3-642-19589-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011923367

CR Subject Classification (1998): H.4, H.3, D.2, K.6, H.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 7th International Workshop on Web Services and Formal Methods (WS-FM 2010), held at the Stevens Institute of Technology, Hoboken, New Jersey, USA, during September 16–17, 2010 and co-located with the 8th International Conference on Business Process Management (BPM 2010). The aim of the WS-FM workshop series is to bring together researchers working on service-oriented computing, cloud computing and formal methods in order to catalyze fruitful collaboration.

This edition of the workshop attracted 26 submissions. We wish to thank all their authors for their interest in WS-FM 2010. After careful discussions, the Programme Committee selected 11 papers for presentation at the workshop. Each of them was accurately refereed by at least three reviewers, who delivered detailed and insightful comments and suggestions. The workshop Chairs warmly thank all the members of the Programme Committee and all their sub-referees for the excellent support they gave, as well as for the friendly and constructive discussions. We also would like to thank the authors for having revised their papers to address the comments and suggestions by the referees.

The workshop programme was enriched by the outstanding invited talks by Rick Hull of IBM Watson Research Center, USA, and Shriram Krishnamurthi of Brown University, USA.

We would like to thank the BPM 2010 organizers for the significant amount of support they provided for the organization of the WS-FM 2010.

We are also grateful to Andrei Voronkov, who allowed us to use the wonderful free conference software system EasyChair, which we used for the electronic submission of papers, the refereeing process and the Programme Committee work.

November 2010

Mario Bravetti
Tevfik Bultan

Organization

Programme Committee Co-chairs

Mario Bravetti	University of Bologna, Italy
Tevfik Bultan	University of California at Santa Barbara, USA

Programme Committee

Wil van der Aalst	Eindhoven University of Technology, The Netherlands
Matteo Baldoni	University of Turin, Italy
Samik Basu	Iowa State University, USA
Karthikeyan Bhargavan	Microsoft Research-INRIA Joint Centre, France
Nicola Dragoni	Technical University of Denmark, Denmark
Marlon Dumas	University of Tartu, Estonia
Schahram Dustdar	Technical University of Vienna, Austria
José Luiz Fiadeiro	University of Leicester, UK
Howard Foster	Imperial College London, UK
Xiang Fu	Hofstra University, USA
Stefania Gnesi	ISTI-CNR, Italy
Sylvain Hallé	Université du Québec à Chicoutimi, Canada
Thomas Hildebrandt	IT University of Copenhagen, Denmark
Kohei Honda	Queen Mary, University of London, UK
Manuel Mazzara	University of Newcastle, UK
Manuel Núñez	Universidad Complutense de Madrid, Spain
Gwen Salaün	Grenoble INP - INRIA - LIG, France
Jianwen Su	University of California at Santa Barbara, USA
Karsten Wolf	University of Rostock, Germany
Gianluigi Zavattaro	University of Bologna, Italy

Additional Reviewers

César Andrés	Cinzia Di Giusto	Michele Mazzucco
Cristina Baroglio	Gregorio Díaz	Andrew Miner
Andrea Bracciali	Khaled Farj	Meriem Ouederni
M. Emilia Cambronero	Ilaria Matteucci	G. Michele Pinna

Table of Contents

Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles

Richard Hull[1,⋆], Elio Damaggio[2,⋆⋆], Fabiana Fournier[3,⋆⋆⋆], Manmohan Gupta[4], Fenno (Terry) Heath III[1], Stacy Hobson[1], Mark Linehan[1], Sridhar Maradugu[1], Anil Nigam[1], Piyawadee Sukaviriya[1], and Roman Vaculin[1]

[1] IBM T.J. Watson Research Center, USA
{hull,theath,stacypre,mlinehan,
sridharm,anigam,noi,vaculin}@us.ibm.com
[2] University of California, San Diego
damaggio@cs.ucsd.edu
[3] IBM Haifa Research Lab, Israel
fabiana@il.ibm.com
[4] IBM Global Business Services, India
manmohan.gupta@in.ibm.com

Abstract. A promising approach to managing business operations is based on *business entities with lifecycles* (*BEL's*) (a.k.a. *business artifacts*), i.e., key conceptual entities that are central to guiding the operations of a business, and whose content changes as they move through those operations. A BEL type includes both an *information model* that captures, in either materialized or virtual form, all of the business-relevant data about entities of that type, and a *lifecycle model*, that specifies the possible ways an entity of that type might progress through the business by responding to events and invoking services, including human activities. Most previous work on BEL's has focused on the use of lifecycle models based on variants of finite state machines. This paper introduces the *Guard-Stage-Milestone* (*GSM*) meta-model for lifecycles, which is an evolution of the previous work on BEL's. GSM lifecycles are substantially more declarative than the finite state machine variants, and support hierarchy and parallelism within a single entity instance. The GSM operational semantics are based on a form of Event-Condition-Action (ECA) rules, and provide a basis for formal verification and reasoning. This paper provides an informal, preliminary introduction to the GSM approach, and briefly overviews selected research directions.

Keywords: Business Artifact, Business Entity with Lifecycle, Business Operations Management, Business Process Management, Case Management, Data-centric Workflow, Declarative.

1 Introduction

The trend with business activities is towards increased complexity, globalization, and out-sourcing. At an infrastructure level, the use of SOA, of Software as a Service (SaaS), and cloud-based computing will become increasingly important. At a semantically richer

⋆ This author partially supported by NSF grant IIS-0812578.
⋆⋆ Research by this author performed while employed as a summer Ph.D. intern at IBM T.J. Watson Research Center.
⋆⋆⋆ This author partially supported by EU Project FP7-ICT ACSI (257593).

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 1–24, 2011.

level, the framework and tools for structuring and managing future business operations and processes will evolve from technologies including workflow, Business Process Management (BPM), case management, and document engineering. A particularly promising approach for managing business operations and processes, especially those that cut across silos and organizations, is based on[1] *Business Entities with Lifecycles* (*BEL's*) [23,20,27,4,8], and related works such as [2,6,26,30]. This paper introduces an evolution of that work, called *Business Entities with Guard-Stage-Milestone Lifecycles*, abbreviated as *BEL[GSM]* or simply *GSM*, in which the lifecycles for BEL's are much more declarative than previous work. The paper describes a preliminary version of the GSM meta-model[2] and framework, highlights some of its novel features, discusses how it can provide support for business-level stakeholders in the design process, and lists some verification and reasoning problems raised by the approach.

The BEL framework is focused on enabling the design, deployment, and ongoing use of *Business Operations Models* (BOMs) that manage a business-meaningful scope of a business. BEL's are key conceptual entities that are central to the operation of part of a business and that change as they move through the business's operations. A BEL type includes both an *information model* that uses attribute/value pairs to capture, in either materialized or virtual form, all of the business-relevant data about entities of that type, and a *lifecycle model*, that specifies the possible ways that an entity of this type might progress through the business, and the ways that it will respond to events and invoke external services, including human activities. An example BEL type is *financial deal*, including evaluation of customer credit and collateral, negotiating interest and terms, tracking of payments, etc. (see [8]). The BEL or *entity-centric* approach is said to be "data centric", because of the emphasis on information and how it evolves. This is similar to case management (e.g., [30,13]), and contrasts sharply with process-centric approaches, such as BPMN [25], where modeling the data manipulated by a Business Process is performed largely outside of the process specification. The entity-centric approach enables business insights and improves communication among diverse stakeholders about the operations and processes of a business, in ways that activity-flow-based and document-based approaches have not [8,10].

Almost all of the previous work on BEL's is based on meta-models for entity lifecycles that use variants of finite state machines. The GSM meta-model uses a considerably more declarative approach to specifying the lifecycles. The meta-model builds on recent work exploring theoretical aspects of declarative approaches to BEL's (e.g., [5,14,15]), and adds a number of practically motivated features. At the core of GSM lifecycle models is the notion of *stage*, which is based on three main constructs: (i) *milestone*, a business-relevant operational objective expressed using a condition over the information model and possibly a triggering event; (ii) *stage body*, containing one or more activities, including calls to external services and possibly sub-stages, intended

[1] In much of the previous research literature on BEL's, the term 'Business Artifact' has been used in place of the term 'Business Entity (with Lifecycle)'. These terms refer to the same concept. Within IBM, the team has been shifting to 'entity', because the term 'artifact' has a different, well-established meaning in the community of IT practitioners in the BPM space.

[2] Following the tradition of UML and related frameworks, we use here the terms 'meta-model' and 'model' for concepts that the database and workflow research literature refer to as 'model' and 'schema', respectively.

to achieve a milestone (or one of several related milestones); and (iii) *guard*, a condition and possibly a triggering event that, when achieved, enables entry into the stage body. The conditions for milestones and guards range over the information model of the BEL instance under consideration, and possibly those of related BEL instances. The conditions are expressed using (a modest extension of) OMG's Object Constraint Language (OCL) [16], which supports the classical first-order logic constructs but adapted to nested data structures. The triggering events for milestones and guards, if present, might be from the external world (e.g., from a human performer or an incoming service call), might be from another BEL instance, or might be the result of milestones changing values or stages changing their status. Hierarchy is achieved through nesting of stages, and multiple stage bodies of a single BEL instance may run in parallel. The operational semantics of GSM is given by Event-Condition-Action (ECA) rules that are derived from the guards and milestones. Indeed, GSM can be viewed as an approach to structuring ECA rules in the BPM context. We hypothesize that because GSM has a basis that is much more declarative than, e.g., BPMN and earlier BEL meta-models, it will be easier to specify variations (as arise, e.g., due to regional differences) and to modify GSM models to reflect changing business requirements.

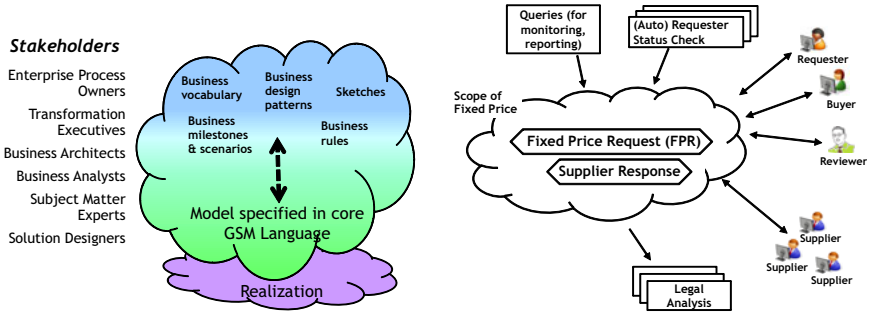
This short paper is intended to provide a preliminary introduction to the GSM meta-model and framework. Design extensions and refinements of GSM are still underway; these are now guided largely by using GSM to specify various application scenarios drawn from past and current engagements. Being somewhat preliminary, the version of GSM described here can be considered to be "Version 0.9". A detailed description of the "Version 1.0" of GSM is currently in preparation [18], along with a first paper on formal aspects of it [12]. The GSM approach to specifying business entity lifecycles forms part of a larger effort at IBM Research, called Project ArtiFact$^{\mathrm{TM}}$.

Section 2 provides an overview of Project ArtiFact and GSM's role within it. The section also introduces a running example for the paper, and illustrates how software based on GSM can be integrated into a larger environment with SOA components and human performers. Section 3 describes the core GSM meta-model. Section 4 overviews the operational semantics of GSM. A short survey of verification and reasoning problems raised by GSM and Project ArtiFact is presented in Section 5. Related research is discussed in Section 6 and brief conclusions offered in Section 7.

2 Overview of Project ArtiFact

Project ArtiFact is exploring several broad themes in the BPM space. The overarching goal is to support the wide range of stakeholders involved with specifying and managing business operations. Figure 1a lists the main categories of such stakeholders, with the exclusion of IT staff. At one end, the stakeholders include *Enterprise Process Owners* and *Transformation Executives*, who have strong intuitions and want to understand the BOM at a very coarse-grained level, and at the other *Solution Designers*, who will create a specification of the BOM in enough detail that it can be translated into a running system. The stakeholders in between these extremes will be interested in understanding and specifying elements of the BOM at varying levels of detail. We refer to all of the stakeholders above the Solution Designers as *business-level stakeholders*.

A central vision of Project ArtiFact is to enable the various business-level stakeholders to create and work with "BOMs" that are intuitive, inprecise, and/or incomplete, and yet be able to map these into GSM specifications. For example, we want to enable business-level stakeholders to informally specify a family of entity types and some of their key milestones, and then specify a number of business scenarios using the milestones (e.g., corresponding to the "sunny day", to working with customers with credit risk, etc.). The transformation from these scenarios to GSM specifications would be based on a combination of inputs from various kinds of stakeholders along with automated verification and synthesis algorithms. Business rules, e.g., in the sense of OMG's Semantics of Business Vocabulary and Rules (SBVR) [24], will also play an important role in the support of business-level stakeholders. Conditions and rules become essential when business policies are too intricate or cumbersome to express in a graphical format alone. The core constructs of GSM were chosen on the one hand to be very close to the ways that the business stakeholders think – in terms of milestones and business rules – and on the other to enable both a formal foundation and a fairly direct mapping of GSM BOMs into running systems.



(a) Various stakeholders and perspectives on a Business Operations Model

(b) Scope of fixed-price contracting scenario

Fig. 1. Contexts when using GSM

A second broad goal of Project ArtiFact is that the GSM BOMs, while reasonably intuitive, are *actionable*, in the sense that there is a relatively direct path from the specification to an implementation of running systems. As demonstrated by the previous work on Business Entities with Lifecycles, currently embodied in IBM's BELA offering [27], having an actionable meta-model enables the creation of a rapid prototype of a BOM after just two or three days of modeling, and substantially reduces the risk and time involved with mapping a detailed BOM into an IT realization.

A third broad goal of GSM is to enable seamless support for highly "prescriptive" approaches to managing business operations (e.g., as embodied in standards such as BPMN), and highly "descriptive" approaches (e.g., as embodied in the *ad hoc* style of activity management found in case management systems [13,30]). It appears that many constructs found in classical process-centric approaches to BPM can be simulated in GSM, and it is natural to include such constructs into GSM as macros (see Subsection 3.9). There is also a close correspondence between GSM and case management, and as briefly discussed in Section 6, it appears that GSM can simulate the traditional case

management constructs. GSM may provide a useful platform for exploring new ways to blend constructs from the prescriptive and descriptive approaches.

The GSM meta-model is being developed and studied in several ways. A concise, text-based programming language, called *GSM-L*, is being designed. A prototype engine, called Barcelona, is being developed to support experiments and implementations using GSM. (This supports a simple graphical design editor, and captures the GSM BOMs directly into an XML format.) Barcelona is an outgrowth of the Siena system [9], that supports a BEL meta-model with state-machine based lifecycles. Also, GSM is providing the basis for the development of a framework to support intuitive design of BOMs by business-level stakeholders. Finally, a formal specification of the (salient aspects of the) GSM meta-model is being developed and applied [12].

This section concludes by introducing a running example based on a real-world contracting application scenario called *Fixed-Price*. This application automates supply chain management for purchasing services at fixed, predetermined prices. Fixed-Price enables *Requesters* in an enterprise to find and hire external professional services contractors to achieve specified goals. The Requesters submit an initial request to a *Buyer*, who focuses on numerous technical, financial, and legal aspects of the request. If approved by the Buyer, then the request is submitted to multiple *Suppliers* for competitive bidding. The Suppliers may decline to respond, or may submit a response back to the Buyer. The responses are evaluated and a winner chosen. Finally, requisitioning takes place with the winner.

Figure 1b illustrates the scope of Fixed-Price as it might be implemented using BEL's. It is natural to use two entity types, called *Fixed Price Request* (FPR) and *Supplier Response*. An FPR instance is born when the Requester makes an initial draft of the request. The Requester may edit it and then submit to the Buyer to modify and approve it; after a winner is chosen the FPR instance enables management of the requisitioning with the winner. The Supplier Response entity type is used to manage each individual bid from the Suppliers: from initial notification of a Supplier; response by that Supplier; evaluation of the response by the Requester, the Buyer, and also one or more *Reviewers*; and finally managing notifications to the Suppliers about the outcome.

As discussed below, a collection of BEL types can be managed within a software container called a *BEL Service Center* (*BSC*). For example, in Figure 1b, the cloud icon represents a BSC that is managing the two BEL types and all of the currently active instances of those types. In typical usage, the BSC includes SOA interfaces (REST and/or WSDL) to support interactions with external services, and also interfaces so that human performers can interact with the BEL instances, both by performing tasks and by sending messages (i.e., events) to influence the flow of control.

3 The Guard-Stage-Milestone Meta-model

This section introduces the four key constructs of the core GSM meta-model, namely, information models, guards, stage bodies, and milestones. It also briefly considers related aspects of the GSM meta-model, including macros, BEL Service Centers, transactional consistency, the incorporation of human *performers*, and exception handling. The discussion is based largely on the Fixed-Price scenario from Section 2. The section also provides a few illustrations of the extended OCL syntax used for conditions in GSM-L.

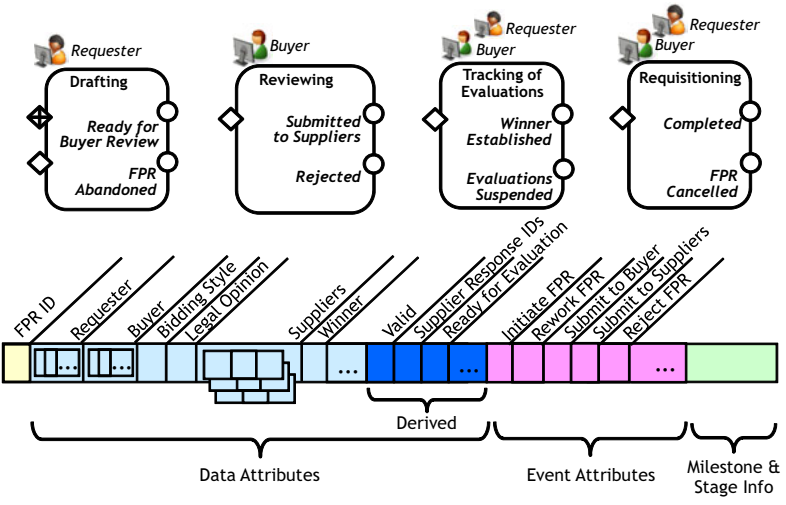


Fig. 2. Information model and top-level stages of FPR entity type

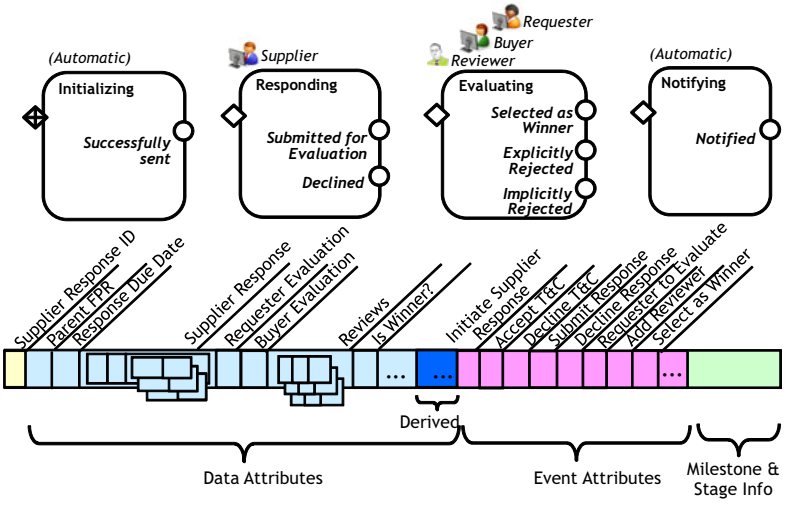


Fig. 3. Information model and top-level stages of Supplier Response entity type

3.1 Information Models

Figures 2 and 3 illustrate the FPR and Supplier Response entity types, respectively, including sketches of their information model and of the top-level stages of their lifecycle models. The information models for the two entity types are illustrated along the bottom halves of the two figures. Following the tradition of previous Business Entity metamodels, a central axiom of the BEL paradigm is that all business-relevant information about an entity instance should be recorded in its information model. This contrasts with typical process-centric approaches, where a substantial amount of business-relevant data

arises in process variables but is difficult or impossible to access from outside the immediate scope of use. The entity information typically includes (i) data provided by human performers; (ii) data about external services that have been called, and (iii) data that essentially holds a log of what has happened so far to the entity instance.

The information model of a GSM entity type will be of record type, where each field is either a scalar, a record type, or a collection type (set, bag, and list), and where the record and collection constructs can be nested arbitrarily.

As suggested in Figures 2 and 3, the attributes of an information model are broken into three categories. The *data attributes* hold business-relevant data about the progress of an entity instance. For FPR, this includes information about the Requester, the Buyer, information about the proposed contract, the list of Suppliers to be considered, the identity of the winning Supplier, information about the requisitioning phase, etc. As discussed in Subsection 3.7 below, *derived* data attributes are also supported.

The *event attributes* hold information about event occurrences of external business-relevant event types that are relevant to a given entity instance. An event type is considered to be *external* (to the pure GSM processing) if (a) it originates from outside of the BSC, (b) it originates as a message from one entity instance to another entity instance, or (c) it corresponds to the termination of a computational task (e.g., an assignment task). A given event occurrence may be relevant to multiple instances, in which case it is conceptually recorded on all of them. As will be discussed below, each such event type may serve as the triggering event type for one or more guards and/or milestones in the lifecycle model of the entity type. External event types specify the structure of payload that is associated with occurrences of the type, including attributes for timestamp and sender. Each event attribute will hold information about the most recent relevant occurrence of the event type, and may optionally hold a history of previous relevant occurrences of the event type. (In the case of events corresponding to service call returns, information in the payload will also be written directly into selected data attributes; see discussion in Subsection 3.6 below.)

The final portion of the information model holds *milestone and stage info*; this will be described in Subsection 3.3 below.

3.2 Overview of Lifecycle Models

Lifecycle models specify the business-relevant activities that an entity instance can be involved in, along with their possible placement in time, as the entity passes through the business operations. A lifecycle model is specified using *stages*, where each stage consists in one or more *milestones*, a *stage body*, and one or more *guards*. Nesting of stages is supported, and also, stages at the same level of nesting may execute in parallel. A stage is *atomic* if it has no substages and *non-atomic* otherwise (see Subsections 3.6 and 3.9).

As will be seen, in GSM an entity instance will move through its lifecycle as the result of events coming from the external world (and possibly from other entity instances in the same BSC). When an external event is processed, it may result in a series of guards becoming true and/or milestones changing value, along with stages becoming open and/or closed. This is captured in the operational semantics using chaining of ECA rules derived from the BOM (see Subsections 3.3 and 4.1).

In the context of GSM, human performers play a central role in guiding the processing, in two distinct ways. First, as is typical of most workflow and Business Process Management systems, they can perform human services that are invoked from within atomic stages. Second, in GSM the human performers can control the flow of processing, by sending events into the BSC that have the effect of triggering guards and milestones, that is, of opening and closing stage occurrences, respectively. GSM will provide a natural framework for incorporating rich, declarative constructs for managing how people can interact with BOMs; this is a direction of current research.

The upper portions of Figures 2 and 3 show the topmost levels of the lifecycle models of FPR and Supplier Response. There are four top-level stages for the FPR type. Each stage has one or more milestones, which correspond to key business-relevant operational objectives that an FPR instance might achieve over its lifetime. Importantly, the milestones used in the lifecycle model for FPR in Figure 2 correspond directly to the ways that managers of the real-world Fixed-Price application speak about the top-level steps to be achieved by each fixed-price request, and similiarly for Figure 3.

The stage Drafting enables a Requester to create an initial fixed-price contracting request. After drafting the request, the Requester can send the event Submit to Buyer, which triggers the Ready for Buyer Review milestone, assuming that certain business-relevant conditions are met (e.g., that certain attribute fields have been filled in, that the end date is after the start date, etc.). Alternatively, the Requester can send the Abandon FPR event, which triggers the FPR Abandoned milestone. The Reviewing stage (see Figure 4) is where the Buyer reviews, edits, and augments the request, so that it can be submitted to the Suppliers. The key milestone here is Submitted to Suppliers (for their responses); this becomes true when the call-for-bids has been submitted to the

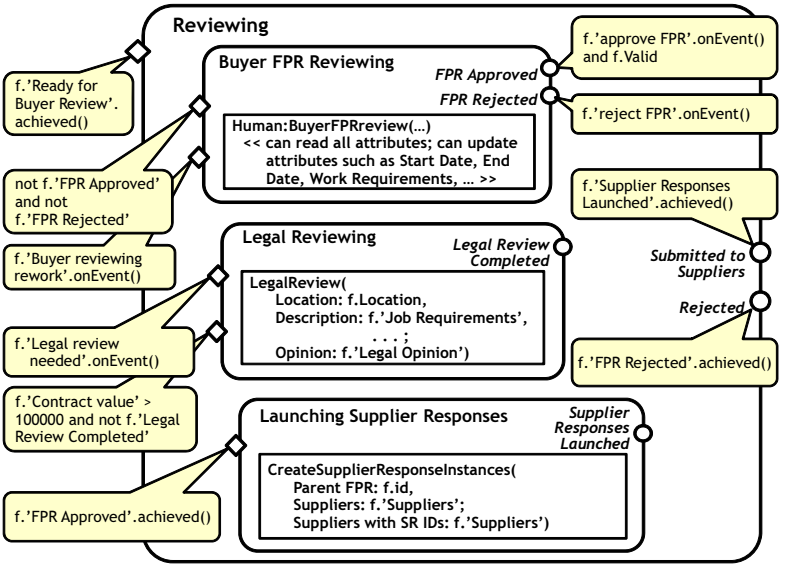


Fig. 4. Drill-down into Reviewing stage of FPR entity type

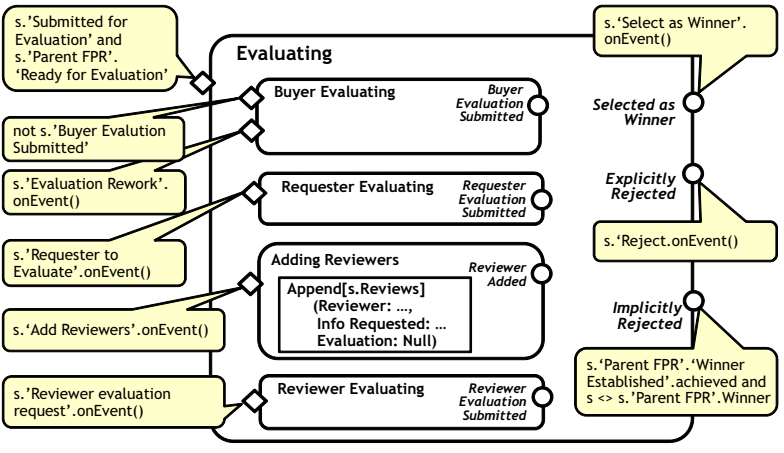


Fig. 5. Drill-down into Evaluating stage of Supplier Response entity type

Suppliers. (Or to be technically correct, when the Supplier Response entity instances have been created; those have an automated step to notify the Suppliers by email.)

As will be seen shortly, the bulk of the evaluation of supplier responses is modeled as part of the lifecycle of the Supplier Response entity type. However, some bookkeeping tasks are also performed at the level of the FPR instance; this is done in the Tracking of Evaluations stage, whose major milestone is Winner Established. This milestone becomes true when some Supplier Response entity instance has achieved the milestone Selected as Winner, that is, when the Buyer has selected a particular supplier's response to be the winner of the competition.

The final top-level stage of FPR is Requisitioning of a contract with the winning supplier; this is not discussed in the current paper.

The lifecycle model of Supplier Response also has four top-level stages. The first one, Initializing, is automated; it populates selected attributes of the Supplier Response instance and then sends an email to the Supplier. Responding enables the Supplier to respond; he or she must first accept or decline the Terms & Conditions, and if accepted, then fill in various attributes of the information model. By sending an event of type Submit Response the Supplier indicates that the response should be submitted back to the Buyer. The Evaluating stage, which is described in more detail below (see Figure 5), manages the evaluations by Buyer, Requester, and Reviewers. It is from within this stage that the Buyer can indicate that a supplier response is the winner, or explicitly reject it. If some other response to the FPR instance is selected as winner, then the Implicitly Rejected milestone will be be achieved automatically. (See Subection 4.2 for a detailed description of how this occurs in the operational semantics.) Once a decision is made for a supplier response, then the Notifying stage sends an email to the Supplier.

3.3 Stage and Milestone Status

Each milestone m will include a boolean attribute, denoted simply by 'm', which holds the current *status* of m. This is initialized to FALSE, and may become TRUE under

certain circumstances. For example, f.'Ready for Buyer Review' will become true for some FPR instance f if the Requester submits a Submit to Buyer event and certain business-relevant requirements are satisfied. In this case we say that the milestone has been *achieved*. In some cases, a milestone with value TRUE may become compromised or invalidated. For example, in FPR if a Buyer rejects an FPR request submitted by a Buyer, then the milestone Ready for Buyer Review is *invalidated*, and the associated value assigned to FALSE.

During the life of a BEL instance, if a guard of some stage S becomes true, then an *occurrence* of S is launched; in this case we say that S has become *open* (or *active*). If S is open and a milestone of S becomes true, then the occurrence is ended, and we say that S has become *closed* (or *inactive*).

In some cases it is useful to permit the guards and milestone conditions to refer directly to the status of stages and/or milestones, and so a BEL information model holds two kinds of *status attributes*. This includes for each milestone m the status attribute m discussed above. It also includes, for each stage S, a boolean status attribute $active_S$; This is assigned TRUE if some occurrence of S is open, and FALSE otherwise.

In some cases it is useful to use the change in value of a status attribute as a triggering event for some guard or milestone. GSM thus supports four kinds of *status-change event types*. The first two, denoted in the GSM-L syntax as S.opened() and S.closed(), are satisfied at the logical point in time[3] where S is opened and closed, respectively. The other two, denoted m.achieved() and m.invalidated(), are satisfied at the points in time when m toggles to TRUE and to FALSE, respectively.

In a BEL information model, the *milestone and stage info* attributes hold information about how the values associated with stages and milestones change over time. At a minimum, for each stage S (milestone m, resp.) the current value of status attribute $active_S$ (m) is maintained, along with the logical timestamp of when it last changed. Additional information might be stored, e.g., a history of changes to the status attribute values, and perhaps a count of how many times they have toggled.

3.4 Events and Sentries

As discussed above, there are two categories of event types: external event types and status-change event types. The syntax to test for an occurrence of external event type e is e.onEvent(). The syntax for status-change event types was given in Subsection 3.3.

In the current GSM meta-model, both event occurrences and conditions over all attributes in the information models of the BEL types in a BSC can be used when specifying the circumstances under which a stage should open, when a milestone should become true, and when a milestone should be invalidated. We use the term "sentry" to refer to the kind of component that guards and milestones have in common. More specifically, a *sentry* is a condition having either the form '**on** ξ' or the form '**on** ξ **if** φ' or the form '**if** φ', where ξ is an expression for an external or status-change event type and φ is a condition not involving any event occurrence expressions. As will be seen, a guard is simply a sentry; sentries are used to specify when milestones become true; and sentries are used to specify when milestones become false.

[3] The notions of 'logical point in time' and 'logical timestamp' are discussed in Section 4.

In typical cases a sentry refers only to events and attributes of the entity instance under consideration, but a sentry may refer to events and/or attributes of other entity instances. For example, as illustrated in Figure 5, the sentry for milestone Implicitly Rejected for BEL type Supplier Response is **if** s.'Parent FPR'.'Winner Established' and s <> s.'Parent FPR'.Winner. indicating that this milestone should become true when the Winner Established milestone of the FPR instance associated to s becomes true. (The second conjunct is included as a safeguard.) Also, the substage Winner Bookkeeping of FPR's Tracking of Evaluations has guard[4] f.'Supplier Response IDs' -> exists('Selected as Winner'. Here '-> exists' represents a binary predicate whose semantics is, speaking loosely, that the collection of Supplier Response ID's satisfies the condition that for at least one element s, s.'Selected as Winner' is TRUE.

In the current formulation of GSM, a sentry is based on at most one event occurrance; complex event types are not explicitly supported. Such events can be tested for within the current framework, if histories are maintained in the information model for the external events and the status attributes.

Suppose a guard for stage S has form **on** e.onEvent() **if** φ. In core GSM, if an event occurrence of type e is being processed, and φ is not true, then the event occurrence is "dropped" and S is not opened. In some cases it may be desirable to permit S to open if φ becomes true within some duration. This can be supported as a macro, using a syntax such as e.sinceEvent<*duration*> or m.sinceAchieved<*duration*>.

3.5 Milestones

As noted above, milestones correspond to business-relevant operational objectives that might be achieved by an entity instance. At a technical level, a milestone has a *name*, an associated attribute in the BEL information model, a set of one or more *sentries*, and a set of zero or more *invalidating sentries*. When considered as an attribute, milestone m has type Boolean, and the function timestamp() can be used to access the logical timestamp when it last changed.

To illustrate, in FPR there is one sentry for Ready for Buyer Review, which is **on** f.'Submit to Buyer'.onEvent() **if** φ. Here f is a variable that holds the FPR instance under consideration when evaluating this sentry. Also, φ is a condition that captures various business-relevant requirements. In our running example, there is one invalidating sentry for this milestone, which is **on** f.Rejected.achieved().

In the GSM framework milestones play three primary roles: to trigger the closing of a stage; to trigger other guards, milestones, and milestone invalidators; and to enable later testing of whether the milestone has been achieved. The first use of milestones makes intuitive sense, because the purpose of the stage is to achieve one of its operational objectives, i.e., one of its milestones. (We note that the milestones of a stage are required to be disjoint, that is, they must be formulated in such a way that it is not possible that two milestones can become true at exactly the same moment.) The reader may ask: if a stage occurrence is terminated and there are substages inside with active occurrences, what happens to them? In the default case, all occurrences of substages are terminated

[4] In GSM-L and the figures, sentries are written with a slightly different syntax. E.g., a sentry of form **if** φ is written simply as φ, and a sentry of form **on** e.onEvent() **if** φ is written as e.onEvent()and φ.

if the parent closes. This corresponds to the intuition that since the milestone has been achieved, no further work is needed from within the stage.

A milestone, considered as an event type, can trigger other milestones and guards. E.g., the guard of Reviewing in FPR is f.'Ready for Buyer Review'.achieved().

The third role of a milestone is to support long-term testability of whether an operational objective has been achieved (and is still considered valid). For example, the expression s.'Submitted for Evaluation', which arises in the guard of Supplier Response stage Evaluating (see Figure 5), is used to test whether the Supplier Response instance referred to by variable s has achieved the Submitted for Evaluation milestone (and it has not since been invalidated).

3.6 Stage Bodies and Atomic Stages

Stages provide a way to structure the activities related to a BEL instance. As noted above, if the guard of a stage S becomes true, then an occurrence of S is launched, and the stage is said to be *open*. When a stage (occurrence) is open, then substages of that stage are eligible to be opened if their guards become true. If S is open, and a sentry of a milestone of S becomes true, then the stage occurrence (and all nested substage occurrences) become *closed*.

In the version of GSM described here, for a given stage S there can be at most one occurrence of S that is open at a given time. (The team is currently exploring a useful generalization, in which a stage might be applied separately to each element of a collection, in a manner similar to the map operator in functional programming.)

We consider now atomic stages; example non-atomic stages are described in Subsection 3.9. In GSM, all of the "real" work, that is, the writing of attribute values and interacting with the outside world, is performed within atomic stages. Such stages may contain one or more *tasks* of various types, including: (a) assignment of attribute values; (b) invocation of (one-way or two-way) external services; (c) request to send response to an incoming two-way service call; (d) request to send an event (message) to one or more identified entity instance(s); and (e) request to create a new entity instance. *Human services*, that is services that human performers do, are considered as a special case of two-way external services. An atomic stage may include zero or one task from the categories of (b), (c), (d), or (e), and any number of assignment tasks; these are combined using a simple sequencing operator.

Following previous work on BEL's, services in GSM that interact with a BSC can explicitly read and write to data attributes in the information model of BEL instances. In particular, if a 2-way service is invoked, then parameters for the service call are populated directly from data attributes of the calling BEL instance, and parameter values from the service call return are written directly into data attributes of that BEL instance. (See Subsection 3.10.) Analogous remarks hold for tasks of types (c) through (e).

The direct access to data attributes by external services corresponds to the intuition that the BEL information model is shared among the multiple stakeholders involved with the associated BEL type, a design feature that fosters communication across different groups and sub-organizations. This is a deliberate and significant departure from traditional SOA and object-oriented programming, where the internal data structures of a service or object are hidden.

Suppose that an atomic stage S includes a task that invokes a two-way external service. It is assumed that the external service is capable of eventually sending a return message into the BSC, and that the BSC can correlate that message to the calling entity instance. This return message is used to trigger one of the milestones of S, and S is closed. In some cases an atomic stage S will have a milestone m that can be triggered before the service call returns. For example, this might arise if the parent stage closes, or if a manager determines that the budget has run out, and processing should stop. The called service may have been designed to be capable of receiving an *abort* message and responding appropriately. GSM supports the sending of such abort messages.

In practice, the external services, especially those performed by humans, may have richer capabilities to interact with the BSC. For example, a service invoked from atomic stage S might be capable of receiving messages from the BSC while in the middle of processing. (These might be generated as one-way service calls from stages other than S.) Also, a human performing a service may want to invoke a save or save and exit command in the middle of processing, while still keepig the atomic stage open.

3.7 Illustration of Derived Attributes

Derived attributes provide a mechanism to enable modularity and simple re-use of expressions. We briefly illustrate the notion of derived attributes using the attribute Ready for Evaluation of FPR. This attribute is used as part of the guards for FPR's stage Tracking of Evaluations and Suppler Response's stage Evaluating (see Figure 5). The expression defining Ready for Evaluation is given now, and explained below.

```
((f.'Bidding style'=FreeForm and f.'Submitted to Suppliers')
 or
 (f.'Bidding style'=Restrictive and f.'Response due date' < now()
  or f.'Supplier Response IDs' ->
     forall('Submitted for Evaluation' or Declined))
) and not f.'Supplier Response IDs' -> exists('Selected as Winner')
```

In our version of Fixed-Price, there are two *styles* for the bidding: *FreeForm* and *Restricted*. In FreeForm bidding, the evaluation of a supplier response can begin as soon as it arrives from a Supplier; in Restricted bidding, no supplier response can be evaluated until either (a) the due date for responses has passed or (b) for each supplier associated to the FPR instance, either the supplier has submitted a response or has declined to respond. The first line indicates that if the bidding style is FreeForm, then the supplier response is eligible for evaluation as soon as the milestone Submitted to Suppliers has been achieved by the FPR instance. (Of course, an individual Supplier Response instance cannot be evaluated until it has been filled out and submitted.)

The second and third lines express the condition for the case of Closed bidding. Here the '-> forall' construct in the third line is a binary predicate which is true if each s in f.'Supplier Response IDs' satisfies s.'Submitted for Evaluation' or s.Declined.

Intuitively, the final conjunct is included in Ready for Evaluation because if a winner has been selected, there is no reason to start any further evaluations. At a technical level, including this conjunct impacts the guards of Tracking of Evaluations and Evaluating, and helps to prevent these stages from opening in cases where a winner has already been selected. (See also Subsection 3.14.)

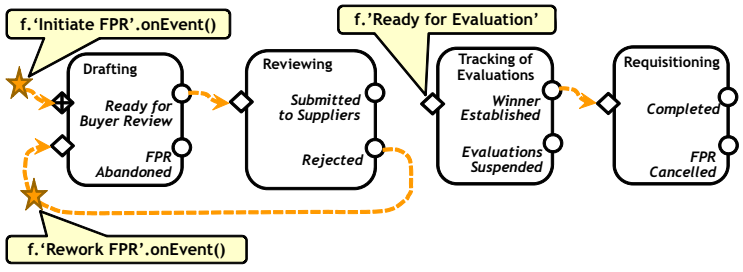


Fig. 6. Illustration of flowchart macros using FRP lifecycle model

3.8 Guards

A guard is simply a sentry associated with a stage. If the sentry becomes true, then (an occurrence of) the associated stage is opened. The guards of a stage are intended to be disjoint, analogous to the milestones of a stage.

Some guards have the special role of launching an entity instance. The upper guards of Drafting in FRP and Initializing in Supplier Response have this role, as indicated by the diamond icon with a cross in it. The sentries of such guards necessarily have an event, and the payload of the event is used to initialize some data attributes of the newly created instance. Also, if such an event is processed successfully, then the BSC sends a return message to the producer of the event, whose payload includes the ID of the newly created instance.

Unlike milestones, guards are not required to have explicit names. This is because business-level stakeholders do not typically refer to guards.

3.9 Macros for Activity Flow Constructs

So far, we have been describing the constructs of the core GSM Language; macros can also be incorporated. Figure 6 illustrates how the semantics of several of the guards and milestones of FPR's lifecycle might be specified using macros based on flowchart constructs. The arrow from Ready for Buyer Review to the guard of Reviewing can be interpreted to mean that the guard is specified as f.'Ready for Buyer Review'.achieved(). The arrow from Rejected to the lower guard of Drafting includes a star, indicating that an event of type Rework FPR is needed in addition to the boolean attribute f.Rejected being true. An open exercise is to see how much of BPMN and other process-centric approachs can be simulated using GSM.

In some cases on can infer flowchart and other constructs from a BOM specified in core GSM-L. This would be helpful during the design process. Exploring efficient ways to perform this kind of inference is an open research challenge for GSM.

3.10 Sub-stages

Figures 4 and 5 are now used to illustrate several key properties of stages and sub-stages. The first shows the FPR stage Reviewing. In the guards and milestones here, the variable f is used to denote the entity instance currently under consideration. The

guard on this stage will become true when the Ready for Buyer Review milestone becomes true. The stage has three substages. Each of these is atomic, and when opened has the effect of launching a (human or automated) external service. For example, Legal Reviewing, if opened, will call the automated service LegalReview which has input parameters Location and Description (and others) which are populated by the attributes Location and Job Requirements (and others) of the FPR instance being worked on, and which has output parameter Opinion which is written into the Legal Opinion attribute of the FPR instance.

Because of the first guard on Buyer FPR Reviewing, this substage is opened automatically when Reviewing is opened. (The two not conditions prevent the substage from opening automatically a second time; see Subsection 3.14.) This substage may also be opened by a message of type Buyer reviewing rework sent to this entity instance.

Substage Legal Reviewing may open for one of two reasons. The first guard indicates that it will be opened whenever a Legal review needed event is sent to the entity instance (and the substage is not currently open). The second guard indicates that if the Contract value is over $100,000 (and the substage has not yet run), then the stage should open as soon as Evaluating opens.

The substage Launching Supplier Responses is opened once the Buyer has approved the FPR instance. It contains a call to the external service CreateSupplierResponseInstances, which has the effect of sending messages into the BSC calling for the creation of a new Supplier Response entity instance for each supplier in the collection attribute f.Suppliers. The ID of each newly created entity instance is returned to the service, which eventually packs all of the IDs into the output parameter Suppliers with SR IDs and returns that to the FRP instance. (In a future version of GSM, an approach for performing such instance creations using explicit GSM constructs may be developed.)

Figure 5 depicts a drill-down into the Evaluating stage of Supplier Response. Recall that this stages enables evaluation of a Supplier Response instance by various parties, and enables the Buyer to explicitly select the instance as a winner or loser (or allow it to be rejected automatically). The guard for this stage is based on a conjunction, with first conjunct being that the Supplier Response instance has been submitted for evaluation, and the second conjuct stating that the parent FPR instance has derived attribute Ready for Evaluation (described above) being true. This sentry illustrates the value of being able to specify a guard without having to specify all the ways that it might be triggered.

The Evaluating stage has four substages, corresponding to the Buyer doing evaluation, the Requester doing evaluation, the adding of one or more Reviewers, and finally a Reviewer doing evaluation. Each of these are atomic, with the first, second, and fourth calling for human services (details not depicted). The body of Adding Reviewer is an assignment task, that appends the payload of the triggering event to the list s.Reviews.

Substage Buyer Evaluating has two guards. The first will become true as soon as Evaluating opens, and so the substage will open automatically. The second guard is used for cases where the Buyer has already completed an evaluation, but wants to revise it; in this case the Buyer sends an Evaluation Rework event. Following the real-world version of Fixed-Price, Requester Evaluating will open only if and when the Buyer sends in a Requester to Evaluate event. Similarly, Adding Reviewers is triggered whenever an Add Reviewers event is received.

As shown in Figure 5, the Reviewer Evaluating substage can open whenever there is an event of type Reviewer evaluation request (and the substage is not already open). Here, the payload of this request identifies the particular reviewer who will perform the evaluation. (As noted above, a useful feature to be added to GSM will allow for parallel occurrences of Reviewer Evaluating, one for each element in s.Reviewers.)

3.11 BEL Service Centers (BSC's)

As suggested in Figure 1b, the BEL instances in Fixed-Price can interact with the "outside world" in four ways. At an implementation level, the component managing the two business entity types and all of their instances is termed a *BEL Service Center* (*BSC*). In addition to managing the actual entity types and instances, the BSC includes a *shell* that provides various services including interfaces to support the interactions with the external world. Individual entity instances can invoke external web services, such as the *Legal Analysis* service; the BSC provides REST and WSDL interfaces and manages the details of the communication for the entity instance. The BSC supports *ad hoc* queries against the data store that holds the entity instances. The BSC can also receive and process incoming web service calls; in this case the BSC shell analyzes the call and routes it to the appropriate entity instance(s) for handling. Finally, the BSC supports the actual run-time interaction between human performers and the entity instances. In principle, this includes support for performers to execute the human services, to see the status of entity instances, and to send events to them.

BSCs also facilitate some communication between entity instances. As noted above, sentries of one entity instance can refer to the information models of other entity instances. Also, BSCs currently enable one instance to send a "message" to another one (which is treated essentially as an external event). Finally, an entity instance can invoke a query against the full family of entity instances currently managed by the BSC.

In the current design, the BSC acts as an SOA-service, that is, it interacts with the outside world exclusively using service calls (specified using REST and WSDL). An interesting research direction is to develop a richer, BEL-aware form of interface to support the interaction between two or more BSCs.

3.12 Parallelism, Transactional Consistency, and Attribute Status

Stage occurrences can run in parallel, which raises the possibility of two stage occurrences attempting to write to the same data attribute, or one stage occurrence attempting to read a data attribute that another one is in the process of writing. To address these possibilities, a transactional consistency discipline based on read and write locks is followed. For each stage, a *ReadSet* and *WriteSet* is specified, where these are sets of data attributes (or subcomponents of them). In the operational semantics, a stage cannot open if it would lead to a locking inconsistency.

There is a relationship between write locks and the *status* of attributes, and the use of attributes in conditions. Attributes generally start in an *uninitialized* status. When a stage S opens, if attribute A is in the WriteSet of S, then A moves into an *active* status, intuitively because the value of these attributes is subject to change at any time. When S closes, assuming that no ancestor of S has A in its WriteSet, then A moves to the *stable* status. If an ancestor S' includes A in its WriteSet, then A is viewed as stable within

the scope of S' and active outside of that scope. Given an atomic condition involving attribute A, if A has active status, then the condition evaluates to false. (The team is currently exploring circumstances where it makes sense to permit a stage to access an attribute even while it is in active status.) Derived attributes may have status *invalid*, e.g., if their evaluation involves a division by zero.

3.13 Exception Handling

As with any programming paradigm, exceptions can arise during run-time, and have to be addressed. In GSM, the basic approach is to include a *fault* milestone for each stage (not done in examples here). Also, for each stage S the Solution Designer may include a substage S' that is intended to handle faults raised by other substages of S. If no such substage is specified, then the fault is propogated to the fault milestone of S.

Recall that all business-relevant data is represented in the information model of an entity type. In principle, then, any exception can be handled by updating selected attribute values in the information model (including possibly adjusting the Milestone and Stage Info). This is one way that the data-centric approach underlying GSM appears to be easier to work with than traditional process-centric approaches.

3.14 Macros and Design Patterns

The focus of the discussion above has been on the core GSM constructs. In some cases the specifications used to obtain certain patterns of behavior among stages are rather cumbersome. It will thus make sense to design macros that can be layered on top of core GSM-L, to simplify both the creation and understanding of GSM BOMs. For example, some macros to simplify specification of activity flow constructs and of fault recovery would be appropriate. We mention here a few additional patterns that are currently under consideration for inclusion as macros.

As illustrated with the Supplier Response Evaluating stage and elsewhere, some guards have no triggering event. This is useful if there are multiple ways that the stage can be triggered, and the Solution Designer does not want to enumerate them, or if the stage is supposed to open automatically when its parent does. As an example of the latter, the second guard of the Legal Reviewing substage (see Figure 4) is f.'Contract value' $>$ 100000 and not f.'Legal Review Completed'. Why is the second conjunct included here? If this were omitted, then as soon as the first occurrence of Legal Reviewing completed then a second occurrence would open. Two macros would be useful to make specification of such guards less cumbersome. The first would use a syntax such as 'no milestone true'; in a guard of some stage S this would evaluate to true only if each milestone of that stage has value FALSE. The second would be a property specified for a stage, with a syntax such as 'run once within(<*stage name*>)', where the named stage is an ancestor of the stage being qualified. The semantics here would be that the stage could run at most once for each occurrence of the named ancestor.

In some cases, if a stage S has completed but later is re-opened, it will be appropriate to invalidate all milestones of all substages of S. In core GSM this is achieved by including explicit invalidators for each milestone. It would be useful to have a macro, as a property of a stage, that indicates that all the milestones of children (or of descendants) should be invalidated if the stage opens.

A common situation in practice is that an active BEL instance (or a family of them) must be *suspended*, that is, work should stop for these instances, but it might be *resumed* later. When an instance is suspended, it may make sense to close some stages, freeze others, and let others run to completion. If the instance is resumed, some milestones (and their associated stages) might have to be reworked or verified, while others might be permitted to remain true. The team is developing macros to simplify specification of various suspend and resume behaviors.

4 GSM Operational Semantics

This section gives a brief sketch of the GSM operational semantics, first in general terms and then with two examples. (A detailed discussion is given in [18, 12].)

4.1 Transformation into ECA Rules

In GSM, progress along lifecycles by a family of BEL instances is organized into *macro-steps*. Each macro-step corresponds to the processing of a single external event, using a sequence of one or more *micro-steps*. The first micro-step involves writing data into an event attribute and possibly into (non-derived) data attributes, and the subsequent micro-steps involve changing the values of status attributes. Each of the micro-steps except the first is an application of an ECA rule derived from the lifecycle models of the BEL types in the BSC.

At a conceptual level, in the GSM operational semantics each time a macro-step is performed in a BSC it is assumed that there is a transactional lock on the full set of entity instances managed by the BSC. (In practical implementations, of course, parallel processing is permitted, as long as the same semantics is achieved modulo the ordering of processing of external events.) Intuitively, once a macro-step begins at a time t, the micro-steps within that macro-step can be viewed as happening "at the same time", i.e., at time t. In this case, t is referred to as the *logical point in time* when each microstep was performed, and is used as the *logical timestamp* of each status change event that occurs as the result of these micro-steps.

Although space limitations prevent a detailed discussion, we present the basic idea of how ECA rules are derived from a GSM BOM. Some of the rules are derived directly from the guards and milestones. These include the following kinds of rules.

(i) (*Opening a stage:*) if g is the guard of a stage S (with parent S'), then there is a corresponding rule with antecedent g (conjuncted with $active_{S'}$) and consequent $+active_S$. The consequent means that the value of status attribute $active_S$ should be changed to TRUE. (If S has parent S', then incorporating the conjunct $active_{S'}$ reflects that g is eligible for consideration only if the parent stage S' is open),

(ii) (*Setting a milestone and closing its stage:*) if α is a sentry for milestone m of stage S, then there is a corresponding rule with antecedent α conjuncted with $active_S$, and as consequent $+m; -active_S$.

(iii) (*Invalidating a milestone:*) if β is an invalidating sentry for milestone m, then there is a corresponding rule with antecedant β and consequent $-m$ (i.e., make m false).

(iv) (*Launching a BEL instance:*) If g is a guard of root-level stage S that launches new entity instances, then there is a corresponding rule with antecedent g and consequent the creation of a new entity instance, making $active_S$ true for that instance, and generating a request (to the BSC) to send a message to the producer of the triggering event with the ID of the newly created entity instance in its payload.

In addition, there are four forms of default ECA rule, as follows.

(v) (*Closing a substage:*) If S is a stage with substage S', then there is the rule with antecedant S.closed() and consequent. $-active_{S'}$.
(vi) (*Invalidating a milestone of an opening stage:*) If S is a stage with milestone m, then there is a rule with antecedant S.opened() and consequent $-m$.
(vii) (*Launching a task:*) If S is an atomic stage with task (or sequence of tasks) T inside, where T includes a 2-way service, there there is a rule with antecedant S.opened() and consequent generation of a request (to the BSC) to invoke T.
(viii) (*Aborting task:*) If S is an atomic stage with task (or sequence of tasks) T inside, and T can receive abort messages, and m is a milestone of S not triggered by completion of T, then there is a rule with antecedant m.achieved() and consequent generation of a request (to the BSC) to abort T.

Various conditions are placed on the lifecycle model and the application of ECA rules to ensure that processing of micro-steps terminates and satisfies other desirable properties. An acyclicity condition relates to dependency relationships between status attributes, as determined by the ECA rules listed above. (If ReadSets and WriteSets are used in the lifecycle model, then these must also be factored into the acyclicity condition.) The ECA rules must be applied in an order based on a topological sort.

4.2 Examples

We now illustrate how external events are processed by a cluster of entity instances in a BSC. For this illustration we focus on the Supplier Response stage Evaluating (Figure 5) and the FPR stage Tracking of Evaluations. Figure 7 depicts the overall context for the illustration. In particular it shows an FPR instance *FPR1* and the several Supplier Response instances *SR1*, *SR2*, . . . , *SRn* associated to it.

The figure shows a key substage of the FPR stage Tracking of Evaluations, called Winner Bookkeeping. The guard of this stage will become true when one of the associated Supplier Response instances achieves the milestone Selected as Winner. This stage holds an assignment task, which places the ID of the winning Supplier Response instance into the FPR attribute Winner. When that assignment task is complete, the Winner Assigned milestone is achieved. Recall that the assignment task is viewed as "external" to the GSM processing.

The first external event we consider is when the Buyer sends a Select as Winner event to *SR1*. This has the direct impact of writing the event and its payload into the information model of *SR1* (the first micro-step). Next, the milestone Selected as Winner is achieved and its value is set to true, and the FPR stage Reviewing closes (the second micro-step). Then, because the guard of Winner Bookkeeping has become true, that stage opens (the third micro-step). Additional micro-steps may occur in order to close each substage of Evaluating that is open for *SR1*. This ends the macro-step.

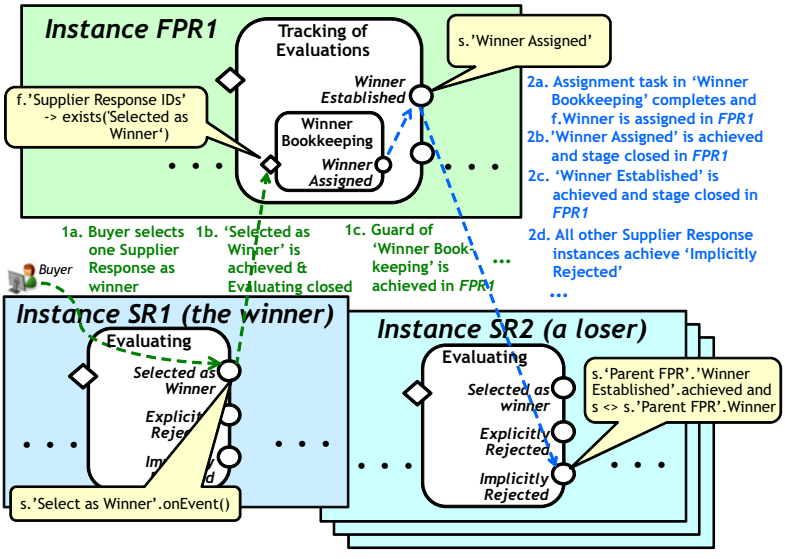


Fig. 7. Illustration of event processing

The second external event occurs when the assignment task in Winner Bookkeeping is completed. The direct impact is that the assigned value is written into f.Winner and the task completion event is recorded into the information model (the first micro-step) This triggers the Winner Assigned milestone and closes Winner Bookkeeping (the second micro-step). This in turn triggers the milestone Winner Established to become true. From here, for each of the Supplier Response instances except *SR1*, the milestone Implicitly Rejected is set (several more micro-steps). Additional micro-steps may occur to close substages of Tracking of Evaluations that are open for *FPR1* and the stages (and substages) of Evaluating for each *SR2*, ... *SRn*. Also, the guard of Requisitioning in *FPR1* will become true because Winner Established became true, and so that stage will open (additional micro-step). The macro-step now ends.

(In the BOM here, the stage Winner Bookkeeping contains only an automated computation, and according to the GSM operational semantics, this task completes before any events from outside of the BSC are processed. If Winner Bookkeeping included a long-running task e.g., to confirm with the winning Supplier that it is prepared to honor the contract, then the BOM would have to be modified to prevent the selection of a second winner before the first winner is fully processed.)

The examples just presented illustrate rather intricate logic at the level of ECA rule chaining. We anticipate that this kind of logic will be understandable in detail by Solution Designers. Importantly, the business-level stakeholders will not need to have a detailed understanding of this kind of logic, although they may develop an intuitive understanding of it. We hypothesize that GSM's ability to capture this kind of detailed logic in a declarative way will simplify the job of modifying BOMs as business requirements change.

5 Representative Verification and Reasoning Problems

This short section identifies several families of well-motivated verification and reasoning problems that can be studied in the context of GSM, and mentions a promising approach that can be used as a starting point to solve them.

As mentioned in Section 2, the Project ArtiFact vision includes enabling business-level stakeholders to specify models of business operations that are intuitive, imprecise, and/or incomplete. For example, these stakeholders will be able to specify a family of high-level business scenarios. It is then natural to ask whether a complete GSM BOM can support this family of scenarios. A much more challenging problem is to synthesize a complete BOM that is intended to support a set of scenarios, and be in some sense minimal among such BOMs. Similar questions can be studied if business rules are used in place of, or along side with, business scenarios.

Another family of questions is raised by the notion of macros on top of the GSM constructs, as discussed in Subsections 3.9 and 3.14. Inferring which macros are implied by a BOM specified using core GSM constructs is very important, because it will help both Solution Designers and business-level stakeholders to understand GSM BOMs.

A third family of questions are classical and deals with issues such as reachability and deadlock. Especially interesting here will be the style of these questions in the context of interactions between BEL instances, if there can be one-many (as in the running example) or many-many relationships between them.

All of these questions are challenging in the GSM context because of the prominance of data, and the resulting infinite state space. While much of the work in the verification community has focused on extending classical model checking to infinite-state systems (see [7] for a survey), in most cases the emphasis is on studying recursive control rather than data, which is either ignored or finitely abstracted. Some recent papers have developed approaches to verification for declarative entity-centric meta-models, including most notably [5, 14, 11]. However, all of these assume that the external services are performed in a serial fashion; parallelism is not supported. An approach has now been developed [12] to reduce verification questions about an abstract GSM BOM into verification questions about BOMs in a meta-model based on the sequential style of the previous results [12]. In particular, given a GSM BOM Γ, it is possible to create a closed-form formula that accurately characterizes the pairs (Σ, Σ') of snapshots of Γ with the property that Σ' can arise as the result of processing a single external event against Σ. From this starting point it is hoped that by using techniques from [14, 11] the verification and reasoning problems mentioned above can be effectively addressed.

6 Related Work

The GSM framework is a synthesis of ideas from several places. The primary influence is the work on state-machine based business entities (artifacts) with lifecycles (see [23, 20, 27, 22], and related work such as [6, 26, 28]). A short survey of that work is [17] Another key influence is the recent theoretical work on incorporating rules-based and declarative constructs into the BEL paradigm (e.g., [5, 14, 15]).

There is a close relationship between the BEL perspective and case management [30, 13]. Both paradigms place heavy emphasis on the data being manipulated by a

business process, and structure the data in terms of a long-lived conceptual entity that can hold all relevant information that is gathered over the lifetime of the entity. Recent work is incorporating into case management rich declarative constructs for managing flow of control; these share several similarities with GSM.

The GSM framework can express most if not all of the constructs typical to case management. To illustrate, we sketch how GSM can support the *to-do list* construct, with which a performer identifies the set of activities deemed necessary for successful completion of a particular case. Let S be a GSM stage with substages $S_1, \ldots, S_n$. To support a to-do list over these substages, begin by including another attribute, called ToDo, which holds a collection over the enumerated set $\{$'S_1', ..., 'S_n'$\}$ (i.e., the names of the substages of S). Also include another substage Modifying ToDo List to S, which enables performers to modify the contents of ToDo. For each guard g of a substage S_i, replace g by g' which is g with an additional conjunct requiring that 'S_i' is present in ToDo. This prevents S_i from opening unless it is in the to-do list. Finally, for each $i \in [1..n]$ a substage Record S_i Completion must be included; this is opened immediately after S_i closes, and deletes 'S_i' from ToDo. Additional details need to be addressed, e.g., to prevent the performer from adding a substage S_i to the to-do list but not some substage S_j that S_i depends on, or at least providing a warning to the performer if this happens. More generally, the ToDo attribute described here could be generalized, e.g., to maintain the history of the substages and other information.

An important influence in the development of GSM is the Vortex workflow model [19], which supports constructs similar to GSM's information model and stages controlled by guards. In Vortex each attribute can be written at most once, and the stages must form a directed acyclic graph; GSM is a substantial generalization over Vortex.

Another influence is Hilda, a declarative language and system for data-driven web applications [31]. While the focus of Hilda is somewhat different, it provides a programming framework based on building blocks that work against a centralized data store using declaratively specified operators. GSM is focused more on managing BEL types rather than arbitrarily structured data stores, and incorporates constructs such as milestone that are intended to model the way that business-level people think.

The AXML Artifact model [2,21] supports a declarative form of BEL's (there called 'artifacts') using Active XML [1] as a basis. The approach takes advantage of the hierarchical nature of the XML data representation used in Active XML. In contrast, GSM uses milestones and hierarchical stages that are guided by business considerations; these might not naturally mimic any particular XML formatting of the underlying data.

DecSerFlow [29] is a fully declarative business process language, in which the possible sequencings of activities are governed entirely by constraints expressed in a temporal logic. GSM does not attempt to support that level of declarativeness. In terms of essential characteristics, GSM can be viewed as a procedural system that permits the use of a rich rules-based paradigm for determining, at any moment in time, what activities should be performed next.

7 Conclusions

This paper introduces a new variant of the Business Entities with Lifecycles (BEL) framework, that uses a declarative approach called Guard-Stage-Milestone (GSM). This

is an evolution of previous work in the BEL paradigm, that has been designed to (a) include constructs that match closely with how business-level stakeholders conceptualize their operations, (b) enable the specification of those constructs in a precise, largely declarative way, and (c) enable a relatively direct mapping from a GSM Business Operations Model (BOM) into a running implementation.

Current research on the GSM meta-model includes (i) creating a framework and tool to support BOM design by business-level stakeholders that is based on specifying business scenarios and other intuitive, imprecise, incomplete specifications; (ii) creating tools to enable the specification of guards, milestones, and derived attributes using an SBVR-inspired structured English; and (iii) developing formal foundations and algorithms for verification and reasoning. Also, the BEL framework, including GSM, forms the basis of a sizeable research effort on a new approach to supporting service interoperation [3]. Additional research directions for GSM include (a) creating a framework support rich collaborative design of GSM BOMs; (b) developing a framework to support variation and change (cf. [21]); (c) finding ways to apply GSM in contexts that include use of legacy business processes and data stores; and (d) finding scalable approaches for supporting large GSM BOMs with massive numbers of entity instances.

Acknowledgements. The authors thank the extended Project ArtiFact team at IBM Research for many informative discussions about the entity-centric approach and its application in business contexts. This includes David Cohn, Opher Etzion, Amit Fisher, Adrian Flatgard, Rong (Emily) Liu, Nir Mashkif, Prabir Nandi, Florian Pinel, Sreeram Ramakrishnan, Guy Sharon, John Vergo, Frederick y Wu. The authors also thank the many people from outside of IBM who have contributed ideas into Project ArtiFact. This includes most notably: Diego Calvanese, Hojjat Ghaderi, Giuseppe De Giacomo, Riccardo De Masellis, Alin Deutsch, Jianwen Su, and Victor Vianu. This also includes members of the EU-funded ACSI research consortium not listed here.

References

1. Abiteboul, S., Benjelloun, O., Milo, T.: The Active XML project: An overview. Very Large Databases Journal 17(5), 1019–1040 (2008)
2. Abiteboul, S., Bourhis, P., Galland, A., Marinoiu, B.: The AXML Artifact Model. In: Proc. 16th Intl. Symp. on Temporal Representation and Reasoning, TIME (2009)
3. Artifact-centric service interoperation (ACSI) web site (2010), `http://acsi-project.eu/`
4. Bhattacharya, K., Caswell, N.S., Kumaran, S., Nigam, A., Wu, F.Y.: Artifact-centered operational modeling: Lessons from customer engagements. IBM Systems Journal 46(4), 703–721 (2007)
5. Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)
6. Born, M., Dörr, F., Weber, I.: User-friendly semantic annotation in business process modeling. In: Weske, M., Hacid, M.-S., Godart, C. (eds.) WISE Workshops 2007. LNCS, vol. 4832, pp. 260–271. Springer, Heidelberg (2007)
7. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification of infinite structures. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 545–623. North-Holland, Amsterdam (2001)
8. Chao, T., et al.: Artifact-based transformation of IBM Global Financing: A case study. In: Intl. Conf. on Business Process Management, BPM (September 2009) (to appear)

9. Cohn, D., Dhoolia, P., (Terry)Heath III, F., Pinel, F., Vergo, J.: Siena: From powerpoint to web app in 5 minutes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 722–723. Springer, Heidelberg (2008)
10. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Engineering Bulletin 32, 3–9 (2009)
11. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic constraints. In: Proc. Intl. Conf. on Database Theory, ICDT (2011) (to appear)
12. Damaggio, E., Hull, R., Vaculin, R.: On the equivalence of incremental and fixpoint semantics for business entities with guard-stage-milestone lifecycles (2011) (in preparation)
13. de Man. H.: Case management: Cordys approach (February 2009), http://www.bptrends.com/deliver_file.cfm?fileType=publication&fileName=02-09-ART-BPTrends%20-%20Case%20Management-DeMan%20-final.doc.pdf
14. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. Intl. Conf. on Database Theory, ICDT (2009)
15. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based workflows. In: Proc. of Intl. Conf. on Database Theory, ICDT (2009)
16. Object Management Group. Object Constraint Language: OMG Available Specification, Version 2.0 (May 2006), http://www.omg.org/technology/documents/formal/ocl.htm
17. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated Intl. Conf.s, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico (2008)
18. Hull, R., et al.: A lifecycle meta-model for business entities based on guards, stages, and milestones (2011) (in preparation)
19. Hull, R., Llirbat, F., Simon, E., Su, J., Dong, G., Kumar, B., Zhou, G.: Declarative workflows that support easy modification and dynamic browsing. In: Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration (1999)
20. Kumaran, S., Nandi, P. (Terry) Heath III, F.F., Bhaskaran, K., Das, R.: ADoc-oriented programming. In: Symp. on Applications and the Internet (SAINT), pp. 334–343 (2003)
21. Marinoiu, B., Abiteboul, S., Bourhis, P., Galland, A.: AXART – Enabling collaborative work with AXML artifacts. Proc. VLDB Endowment 3(2), 1553–1556 (2010)
22. Nandi, P., et al.: Data4BPM, Part 1: Introducing Business Entities and the Business Entity Definition Language (BEDL), (April 2010), http://www.ibm.com/developerworks/websphere/library/techarticles/1004_nandi/1004_nandi.html
23. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
24. Object Management Group (OMG). Semantics of Business Vocabulary and Business Rules (SBVR), Version 1.0 (January 2008), http://www.omg.org/spec/SBVR/1.0/
25. Object Management Group (OMG). Business process management initiative (2011), http://www.bpmn.org/
26. Redding, G., Dumas, M., ter Hofstede, A.H.M., Iordachescu, A.: Modelling flexible processes with business objects. In: Proc. 11th IEEE Intl. Conf. on Commerce and Enterprise Computing, CEC (2009)
27. Strosnider, J.K., Nandi, P., Kumarn, S., Ghosh, S., Arsanjani, A.: Model-driven synthesis of SOA solutions. IBM Systems Journal 47(3), 415–432 (2008)
28. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: A framework for lightweight interacting workflow processes. Int. J. Coop. Inf. Syst. 10(4), 443–481 (2001)
29. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: The Role of Business Processes in Service Oriented Architectures (2006)
30. van der Aalst, W.M.P., Weske, M.: Case handling: a new paradigm for business process support. Data Knowl. Eng. 53(2), 129–162 (2005)
31. Yang, F., Shanmugasundaram, J., Riedewald, M., Gehrke, J.: Hilda: A high-level language for data-drivenweb applications. In: Proc. Intl. Conf. on Data Eng., ICDE (2006)

Simplified Computation and Generalization of the Refined Process Structure Tree

Artem Polyvyanyy[1], Jussi Vanhatalo[2], and Hagen Völzer[3]

[1] Hasso Plattner Institute, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
Artem.Polyvyanyy@hpi.uni-potsdam.de
[2] UBS AG, Postfach, CH-8098 Zurich, Switzerland
jussi.vanhatalo@ieee.org
[3] IBM Research – Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
hvo@zurich.ibm.com

Abstract. A business process is often modeled using some kind of a directed flow graph, which we call a *workflow graph*. The Refined Process Structure Tree (RPST) is a technique for workflow graph parsing, i.e., for discovering the structure of a workflow graph, which has various applications. In this paper, we provide two improvements to the RPST. First, we propose an alternative way to compute the RPST that is simpler than the one developed originally. In particular, the computation reduces to constructing the *tree of the triconnected components* of a workflow graph in the special case when every node has at most one incoming or at most one outgoing edge. Such graphs occur frequently in applications. Secondly, we extend the applicability of the RPST. Originally, the RPST was applicable only to graphs with a single source and single sink such that the completed version of the graph is biconnected. We lift both restrictions. Therefore, the RPST is then applicable to arbitrary directed graphs such that every node is on a path from some source to some sink. This includes graphs with multiple sources and/or sinks and disconnected graphs.

1 Introduction

Companies widely use business process modeling for documenting their operational procedures. Business analysts develop process models by decomposing business scenarios into business activities and defining their logical and temporal dependencies. The models are then utilized for communicating, analyzing, optimizing, and supporting execution of individual business cases within or across companies. Various modeling notations have been proposed. Many of them, for example the Business Process Modeling Notation (BPMN), Event-driven Process Chains (EPC), and UML activity diagrams, are based on *workflow graphs*, which are directed graphs with nodes representing activities or control decisions, and edges specifying temporal dependencies.

A workflow graph can be parsed into a hierarchy of subgraphs with a single entry and single exit. Such a subgraph is a logically independent subworkflow or subprocess of the business process. The result of the parsing procedure is a *parse tree*, which is the containment hierarchy of the subgraphs. The parse tree has various applications, e.g., translation between process languages [1,2,3], control-flow and data-flow analysis [4,5,6,7], process comparison and merging [8], process abstraction [9], process comprehension [10], model layout [11], and pattern application in process modeling [12].

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 25–41, 2011.

Vanhatalo, Völzer, and Koehler [1] proposed a workflow graph parsing technique, called the *Refined Process Structure Tree* (RPST), that has a number of desirable properties: The resulting parse tree is unique and *modular*, where *modular* means that a local change in the workflow graph only results in a local change of the parse tree. Furthermore, it is finer grained than any known alternative approach and it can be computed in linear time. The linear time computation is based on the idea by Tarjan and Valdes [13] to compute a parse tree based on the *triconnected components* of a biconnected graph.

In this paper, we improve the RPST in two ways:

- The original RPST algorithm [1] contains, besides the computation of the triconnected components, a post-processing step that is fairly complex. In this paper, we show that the computation can be considerably simplified by introducing a preprocessing step that splits every node of the workflow graph with more than one incoming and more than one outgoing edge into two nodes. We prove that for the resulting graph, the RPST and the triconnected components coincide. Furthermore, we prove that the RPST of the original graph can then be obtained by a simple postprocessing step. This new approach reduces the implementation effort considerably, requiring only little more than the computation of the triconnected components, of which an implementation is publicly available [14].
- The original technique [1] is restricted to workflow graphs that have a single source and a single sink such that adding an edge from the sink to the source makes the graph biconnected. This assumption is too restrictive in practice as many business process models have multiple sources and/or sinks, some are not biconnected, and some are not even connected. In this paper, we show how these limitations can be overcome. The resulting technique can be applied to any workflow graph such that each node lies on a path from some source to some sink.

The remainder of the paper is structured as follows: The next section defines the RPST and provides additional preliminary definitions. Sect. 3 proposes the simplified algorithm for computing the RPST, and Sect. 4 then generalizes the algorithm to operate on workflow graphs of arbitrary structure.

2 Preliminaries

This section presents the preliminary notions: the RPST [1] in Sect. 2.1, and the triconnected components of the graph [13,15] in Sect. 2.2. We refer to the corresponding original articles for additional motivation of the definitions presented in this section.

2.1 The Refined Process Structure Tree

A *multi-graph* $G = (V, E, \ell)$ consists of two disjoint sets V and E of *nodes* and *edges*, respectively, and a mapping ℓ that assigns to each edge either an ordered pair of nodes, in which case G is a *directed multi-graph*, or an unordered pair of nodes, in which case G is an *undirected multi-graph*. A pair of nodes may be connected by more than one edge (hence the name multi-graph). We assume that the mapping ℓ is fixed, so that a subgraph can be identified with a pair (V', E'), where $V' \subseteq V$ and $E' \subseteq E$ such that each edge in E' connects only nodes in V'. Let $F \subseteq E$ be a set of edges, $G_F = (V_F, F)$ is the subgraph *formed by* F if V_F is the smallest set of nodes such that (V_F, F) is a subgraph.

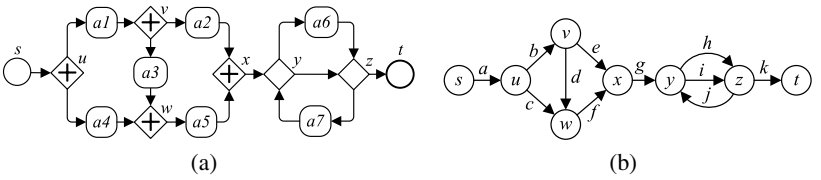


Fig. 1. (a) A workflow graph represented in BPMN, (b) the corresponding TTG (simplified)

A *multi-terminal graph* (*MTG*) is a directed multi-graph G that has at least one source and at least one sink such that each node lies on a path from some source to some sink; G is a *two-terminal graph* (*TTG*) if it has exactly one source and exactly one sink. Fig. 1(a) shows a workflow graph in BPMN notation and Fig. 1(b) presents the corresponding TTG. Note that the activity nodes ($a1, a2$, etc.) are ignored in the TTG for the sake of simplicity. We assume for simplicity of the presentation that a TTG has at least two nodes and two edges.

Let G be an MTG and $G_F = (V_F, F)$ be a connected subgraph of G that is formed by a set F of edges. A node in V_F is *interior* with respect to G_F if it is connected only to nodes in V_F; otherwise it is a *boundary node* of G_F. A boundary node u of G_F is an *entry* of G_F if no incoming edge of u belongs to F or if all outgoing edges of u belong to F. A boundary node v of G_F is an *exit* of G_F if no outgoing edge of v belongs to F or if all incoming edges of v belong to F. F is a *fragment* of a TTG G if G_F has exactly two boundary nodes, one entry and one exit. The set $\{u, v\}$ containing the entry and the exit node is also called the *entry-exit pair* of the fragment. A fragment is *trivial* if it only contains a single edge. Note that every singleton edge forms a fragment. By definition, the source of a TTG is an entry to every fragment it belongs to and the sink of a TTG is an exit from every fragment it belongs to. Intuitively, control 'enters' the TTG through the source and 'exits' the TTG through the sink. Note also that we represent a fragment as a set of edges rather than as a subgraph.

We say that two fragments F, F' are *nested* if $F \subseteq F'$ or $F' \subseteq F$. They are *disjoint* if $F \cap F' = \emptyset$. If they are neither nested nor disjoint, we say that they *overlap*. A fragment of G is said to be *canonical* (or *objective*) if it does not overlap with any other fragment of G. The *Refined Process Structure Tree (RPST)* of G is the set of all canonical fragments of G. It follows that any two canonical fragments are either nested or disjoint and, hence, they form a hierarchy. This hierarchy can be shown as a tree, where the parent of a canonical fragment F is the smallest canonical fragment that contains F. The root of the tree is the entire graph, the leaves are the trivial fragments.

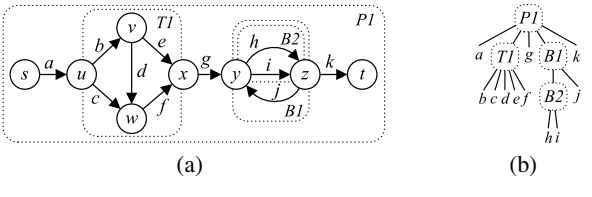


Fig. 2. (a) A TTG and its canonical fragments, (b) the RPST of (a)

Fig.2 exemplifies the RPST. Fig.2(a) shows a TTG and its canonical fragments, where every fragment is formed by edges enclosed in or intersecting an area denoted by the dotted border. For example, the canonical fragment $T1$ is formed by edges $\{b, c, d, e, f\}$, has interior nodes $\{v, w\}$ and boundary nodes $\{u, x\}$, with u being an entry and x an exit of the fragment. Fig.2(b) visualizes the RPST as a tree.

2.2 The Triconnected Components

The fragments of a TTG are closely related to its *triconnected components*, which was pointed out by Tarjan and Valdes [13]. This relationship is crucial for the results that are obtained later in this paper. Here, we introduce the triconnected components in detail and we start with some preliminary definitions.

The *completed version* of a TTG G, denoted $C(G)$, is the undirected graph that results from ignoring the direction of all the edges of G and adding an additional edge between the source and the sink. The additional edge is called the *return edge* of $C(G)$. Let G be an undirected multi-graph. G is *connected* if each pair of nodes is connected by a path; G is *biconnected* if G has no self-loops and if for each triple u, v, x of nodes, there is a path from u to v that does not visit x. If a node x witnesses that G is not biconnected, i.e., there exist nodes u, v such that x is on every path between u and v, then x is called a *separation point* of G. G is *triconnected* if for each quadruple u, v, x, y of nodes, there is a path from u to v that visits neither x nor y. A pair $\{x, y\}$ witnessing that G is not triconnected is called a *separation pair* of G, i.e., there exist nodes u, v such that every path from u to v visits either x or y.

The TTG in Fig.1(b) is connected, but not biconnected; the nodes u, x, y, and z are all separation points. Fig.3(a) shows the completed version $C(G)$ of the TTG from Fig.1(b), where r is the return edge. The completed version is biconnected but not triconnected; $\{u, x\}$ and $\{x, z\}$ are two of many separation pairs of $C(G)$.

Fragments are strongly related to triconnectivity and separation pairs. Note that the entry-exit-pair $\{u, x\}$ of fragment $T1$ in Fig.2(a) is also a separation pair of its completed version in Fig.3(a). In fact, each entry-exit pair of a non-trivial fragment of a TTG G is a separation pair of $C(G)$.

An (undirected) graph that is not connected can be uniquely partitioned into *connected components*, i.e., maximal connected subgraphs. A connected graph that is not biconnected can be uniquely decomposed into *biconnected components*, i.e., maximal biconnected subgraphs. The biconnected components can be obtained by splitting the graph into multiple subgraphs at each separation point. Because of the relationship of

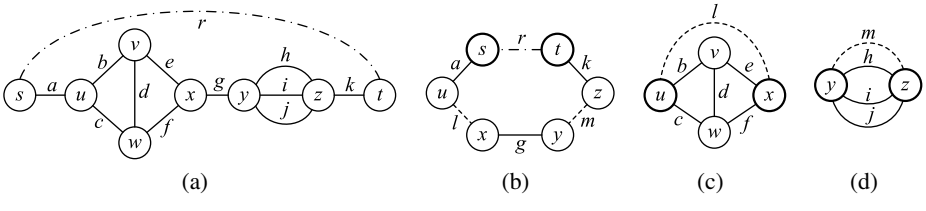


Fig. 3. The completed version of the TTG from Fig.1(b) and its triconnected components: (a) The completed version, (b) a polygon, (c) a rigid component, and (d) a bond

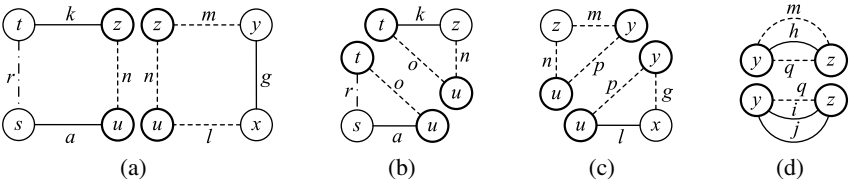


Fig. 4. (a) A split of a hexagon from Fig.3(b), (b)-(c) a split of a tetragon, (d) a split of a bond

fragments to triconnectivity, we are interested to decompose a graph into unique triconnected components. That decomposition is explained in the remainder of this section.

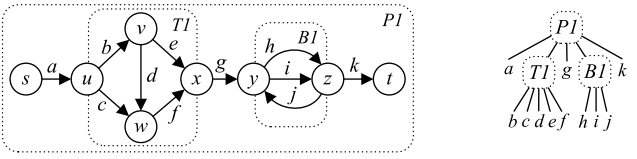
Let G be a biconnected multi-graph and u, v be two nodes of G. A *separation class w.r.t.* u, v is a maximal set S of edges such that any two edges in S are connected by a path that visits neither u nor v except as a start or end point. If there is a partition of all edges of G into two sets E_0, E_1 such that both sets contain more than one edge and each separation class w.r.t. u, v is contained in either of these sets, we call $\{u, v\}$ a *split pair*. We can then *split* the graph into two parts w.r.t. the parameters E_0, E_1 and u, v: To this end, we add a fresh edge e between u and v to the graph, which is called a *virtual edge*. The graphs formed by the sets $E_0 \cup \{e\}$ and $E_1 \cup \{e\}$ are the obtained *split graphs* of the performed split operation. A virtual edge is visualized by a dashed line.

For an example of a split operation, consider the hexagon in Fig.3(b). Note that it already contains virtual edges, which are the result of previous splits. The hexagon can be split along the split pair u, z using the sets $E_1 = \{k, r, a\}, E_2 = \{m, g, l\}$. This results in two tetragons, which are shown in Fig.4(a).

It may be possible to split the obtained split graphs further, i.e., into smaller split graphs, possibly w.r.t. a different split pair. A split graph is called a *split component* if it cannot be split further. Special split graphs are *polygons* and *bonds*. A *polygon* is a graph that has $k \geq 3$ nodes and k edges such that all nodes and edges are contained in a cycle, cf., Fig.3(b). A *bond* consists of 2 nodes and $k \geq 2$ edges between them, cf., Fig.3(d). Each split component is either a *triangle*, i.e., a polygon with three nodes, a *triple bond*, i.e., a bond with three edges, or a *simple* triconnected graph, where *simple* means that no pair of nodes is connected by more than one edge [15]. If a split component is the latter, we also call it a *rigid* component. Fig.3(c) shows an example of a rigid component, whereas the split graphs shown in Fig.3(b) and Fig.3(d) are not split components as they can be split further.

The set of split components that can be derived from a biconnected multi-graph is not unique. To see that, we consider polygons and bonds. For instance, a tetragon, cf., Fig.4(a), can be split along a diagonal into two split graphs. Depending on the choice of the diagonal, two different sets of split components are obtained. Fig.4(b) shows one of the two possibilities for splitting the tetragon given on the left in Fig.4(a). Similarly, a bond with more than three edges, cf., Fig.3(d), can be split into two bonds in several ways, depending on the choice of E_1 and E_2. One possibility to split the bond from Fig.3(d) is shown in Fig.4(d). A set of split components for the graph in Fig.3(a) is given by the graphs in Figs.3(c), 4(b), 4(c), and 4(d).

The inverse of a split operation is called a *merge* operation. Two split graphs formed by edges E_0 and E_1, respectively, that share a virtual edge e between a pair u, v of nodes can be merged, which results in the graph formed by the set $(E_0 \cup E_1) \setminus \{e\}$



(a) (b)

Fig. 5. (a) A TTG and its triconnected component subgraphs, (b) the tree of the triconnected components of (a)

of edges. If we start with a set of split components of G and then iteratively merge a polygon with a polygon and a bond with a bond until no more such merging is possible, we obtain the unique *triconnected components* of G. Because a merge operation is the inverse of a split operation, we can also obtain the triconnected components by suitable split operations only: Let $\mathscr{C}$ be a *split graph decomposition* of G, i.e., a set of split graphs recursively derived from G. A polygon $P \in \mathscr{C}$ is *maximal* w.r.t. $\mathscr{C}$ if there is no other polygon in $\mathscr{C}$ that shares a virtual edge with P. A bond $B \in \mathscr{C}$ is *maximal* w.r.t. $\mathscr{C}$ if there is no other bond in $\mathscr{C}$ that shares a virtual edge with B. $\mathscr{C}$ is a set of the *triconnected components* of G if each member of $\mathscr{C}$ is either a maximal polygon, a maximal bond, or a rigid split component. The set of the triconnected components of G exists and is unique, cf., [15].

The graphs in Fig.4(c) can be merged along the virtual edge p. The obtained tetragon can be merged with the triangles in Fig.4(b) along the virtual edges n and o to obtain the maximal polygon from Fig.3(b). Figs.3(b), 3(c), and 3(d) show all the triconnected components of the graph from Fig.3(a); Fig.3(d) is a maximal bond, which is obtained by merging the bonds in Fig.4(d), and Fig.3(c) is a rigid component.

Any split graph decomposition can be arranged in a tree: The tree nodes are the split graphs. Two split graphs are connected in a tree if they share a virtual edge. The root of the tree is the split graph that contains the return edge. The *tree of the triconnected components* of G is the tree derived in this way from its triconnected components.

Let C be a triconnected component of graph G. Let F be the set of all edges of G that appear in C or some descendant of C in the tree of the triconnected components. The graph formed by F is called the *triconnected component subgraph* derived from C.

Fig.5 shows the tree of the triconnected components. In Fig.5(a), the triconnected component subgraphs of the workflow graph are visualized; they correspond to the triconnected components from Fig.3. Each triconnected component subgraph is formed by edges enclosed in or intersecting a region with the dotted border, e.g., all the graph edges for $P1$ are derived from the component given in Fig.3(b). Fig.5(b) arranges the triconnected components in a tree. The root of the tree, i.e., node $P1$, corresponds to the triconnected component that contains the return edge r. Note the difference between the tree of the triconnected components in Fig.5 and the RPST in Fig.2.

3 Simplified Computation of the Refined Process Structure Tree

In this section, we show how the RPST computation can be simplified compared with the original algorithm. In Sect. 3.1, we discuss the RPST of TTGs in which every node

has at most one incoming or at most one outgoing edge. Such TTGs are common in practice. In Sect. 3.2, we address the general case of the RPST computation of any TTG whose completed version is biconnected.

3.1 The RPST of Normalized TTGs

We call a TTG *normalized* if every node has at most one incoming or at most one outgoing edge. In this section, we show that for normalized TTGs, the RPST computation reduces to computing the tree of the triconnected components. In other words, each canonical fragment corresponds to a triconnected component subgraph and each triconnected component subgraph corresponds to a canonical fragment.

Let $C(G)$ be the completed version of a TTG. A pair $\{x, y\}$ of nodes is called a *boundary pair* if there are at least two separation classes w.r.t. $\{x, y\}$. A separation class is *proper* if it does not contain the return edge. The boundary pair $\{u, x\}$ in Fig. 3(a) generates two separation classes. The first contains the edges b, c, d, e, f and is therefore proper, whereas the second contains all other edges of the graph and is therefore not proper. Fragments are strongly related to proper separation classes. To describe that relationship, we introduce the notion of a *separation component*.

Definition 1 (Separation component). Let $\{x, y\}$ be a boundary pair of $C(G)$. A *separation component* w.r.t. $\{x, y\}$ is the union of one or more proper separation classes w.r.t. $\{x, y\}$.

The bond from Fig. 3(d) without the virtual edge m is a separation component w.r.t. $\{y, z\}$ of the completed version of the TTG from Fig. 3(a). It is the union of the three proper separation classes: $\{h\}$, $\{i\}$, and $\{j\}$.

We know that the entry-exit pair $\{x, y\}$ of a fragment is a boundary pair of G and that the fragment is a *separation component* w.r.t. $\{x, y\}$ [1]. Furthermore, it follows from the construction of the triconnected components that each triconnected component subgraph is a separation component. Polyvyanyy et al. [9] observed that every triconnected component subgraph of a normalized TTG is a fragment. For normalized TTGs, we can extend this observation to a full characterization of fragments in terms of separation components.

Lemma 1. *Let F be a set of edges of a normalized TTG. F is a separation component if and only if F is a fragment.*

Proof. For ($\Rightarrow$), let $\{u, v\}$ be the boundary pair of F and let e be an edge in F. As the return edge is not in F, it is in a different separation class w.r.t. $\{u, v\}$ than e. Consider a simple directed path from the source to the sink of the graph that contains e. It follows that the path contains one of the nodes $\{u, v\}$ before e and one after e; otherwise the separation class of e would contain the return edge. Let, without loss of generality, u be the former node and v the latter. It follows that u has an incoming edge outside F and an outgoing edge inside F, and v has an incoming edge inside F and an outgoing edge outside F. Based on the assumption that the TTG is normalized, it is now straightforward to establish that u is an entry and v is an exit of F. Furthermore, there is no other boundary node besides u and v because that would contradict the definition of a separation class. Hence, F is a fragment.

The direction ($\Leftarrow$) is Theorem 2 in [1]. □

It turns out that the set of triconnected component subgraphs of a normalized TTG is exactly the set of all its canonical fragments and, thus, is the RPST of the TTG. Before we prove the statement, we give two auxiliary lemmas which also by themselves deliver interesting insights into separation components of a normalized TTG and their relations.

Lemma 2. *If F is a separation component and F' a triconnected component subgraph, then F and F' do not overlap.*

Proof. If F contains only a single edge or the entire graph, the claim is trivial. Otherwise F can be split off from the main graph into a split graph. We continue the decomposition until we reach a set of split components. Those can be arranged in a tree (of split components) as described above. F corresponds to a subgraph of this tree, i.e., a subtree represents exactly the edges of F. On the other hand, F' also corresponds to a subtree of the tree of split components because the triconnected components are obtained by merging split components, i.e., by collapsing parts of the tree of split components. Since F and F' both correspond to subtrees of the same tree, they do not overlap. □

It follows from Lemma 2 that triconnected component subgraphs do not overlap. We show now that for a separation component which is strictly contained in a triconnected component subgraph, there always exists another separation component contained in the same triconnected component subgraph that overlaps with it.

Lemma 3. *If F is a separation component that is not a triconnected component subgraph, then there exists a separation component F', such that F and F' overlap.*

Proof. Consider a split graph decomposition that contains F. If F is not a triconnected component subgraph, then F and the parent of F are either bonds w.r.t. the same boundary pair or polygons. In both cases, it is easy to display a bond or polygon, respectively, that overlaps with F. □

We are now ready to prove the main proposition of this section.

Theorem 1. *Let F be a set of edges of a normalized TTG. F is a canonical fragment if and only if F is a triconnected component subgraph.*

Proof.

$\Rightarrow$ Let F be a canonical fragment. We want to show that F is a triconnected component subgraph. Because of Lemma 1, F is a separation component. If F is not a triconnected component subgraph, then there exists, because of Lemma 3, a separation component F' that overlaps with F. Because of Lemma 1, F' is a fragment, which contradicts F being canonical.

$\Leftarrow$ Let F be a triconnected component subgraph. We want to show that F is a canonical fragment. Because of Lemma 1, F is a fragment. Let F' be any fragment. Because of Lemma 1, F' is a separation component. Because of Lemma 2, F and F' do not overlap. Hence, F is a canonical fragment. □

For normalized TTGs, Theorem 1 implies that the tree of the triconnected components and the RPST coincide, i.e., both deliver the same decomposition on the set of edges of the TTG. Fig. 6(a) shows a normalized TTG and its triconnected component subgraphs. The TTG is formed by a subset of edges of the workflow graph from Fig. 1(b). The triconnected component subgraphs are also all the canonical fragments of the TTG. Therefore, the RPST of the workflow graph from Fig. 6(a), which is given in Fig. 6(b), can be computed by constructing the tree of the triconnected components of the workflow graph.

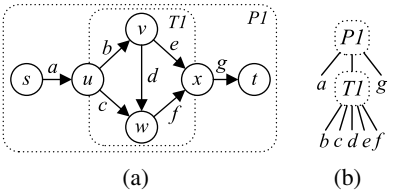


Fig. 6. (a) A TTG and its triconnected component subgraphs, (b) the RPST of (a)

3.2 The RPST of General TTGs

We now show how to compute the RPST of an arbitrary TTG whose completed version is biconnected. To do so, we normalize the TTG by splitting nodes that have more than one incoming and more than one outgoing edge into two nodes. We then compute the RPST of the normalized TTG as in Sect. 3.1. Finally, we project the RPST of the normalized TTG onto the original one and obtain the RPST of the original TTG.

A single node-splitting is sketched in Fig. 7(a). For instance, if the splitting is applied to node u of the graph from Fig. 7(b), it results in the new graph given in Fig. 7(c) with three fresh elements: nodes $*u$ and $u*$, and edge e. This is the only applicable splitting in the example. Hence, the resulting graph is normalized and we call it the *normalized version* of the TTG. The procedure can be formalized as follows.

Definition 2 (Node-splitting). Let $G = (V, E, \ell)$ be a directed multi-graph and $x \in V$ a node of G. A *splitting* of x is *applicable* if x has more than one incoming and more than one outgoing edge. The application results in a graph $G' = (V', E', \ell')$, where $V' = (V \setminus \{x\}) \cup \{*x, x*\}$, $E' = E \cup \{e\}$, where $*x$ and $x*$ are fresh nodes and e is a fresh edge, and ℓ' is such that $\ell'(e) = (*x, x*)$. In addition, $f \in E, \ell(f) = (y, z)$ and $\ell'(f) = (y', z')$ implies that $y' = x*$ if $y = x$, and otherwise $y' = y$; and $z' = *x$ if $z = x$, and otherwise $z' = z$.

Splitting is applicable if and only if the graph is not normalized. It is not difficult to see that the order of different splittings does not influence the final result and, therefore, we indeed get a normal form by applying all applicable splittings in any order.

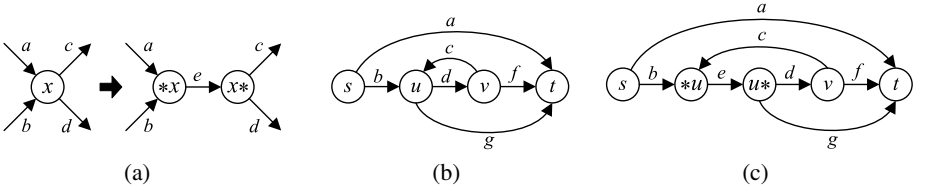


Fig. 7. (a) Node-splitting, (b) a TTG, and (c) the normalized version of (b)

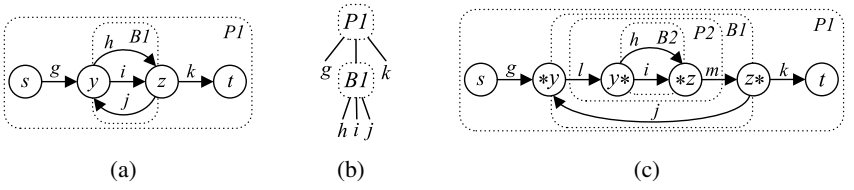


Fig. 8. (a) A TTG and its triconnected component subgraphs, (b) the tree of the triconnected components of (a), and (c) the normalized version of (a) and its triconnected component subgraphs

After normalization, we proceed by computing the tree of the triconnected components of the graph. As we know from Sect. 3.1, the tree coincides with the RPST of the normalized graph. This tree can be projected onto the original graph by deleting all the edges introduced during node-splittings. We will see later that this projection preserves the fragments. However,the deletion of the edges may result in fragments which have a single child fragment. This means that two different fragments of the normalized graph project onto the same fragment of the original graph. We thus clean the tree by deleting redundant occurrences of such fragments. Consequently, the only child fragment of a redundant fragment becomes a child of the parent of the redundant fragment, or the root of the tree if the redundant fragment has no parent. The result is the RPST of the original graph. Alg. 1 details again the sequence of these steps.

Algorithm 1. Simplified computation of the RPST

RPST(Directed multi-graph $G = (V, E, \ell)$**)**
1. $G' = (V', E', \ell')$ is the normalized version of G
2. T' is the tree of the triconnected components of G'
3. T is T' without trivial fragments in $E' \setminus E$
4. R is T without redundant fragments
5. **return** R // `the RPST of` G

We exemplify Alg. 1 in Fig.8 and Fig.9 by computing the RPST of the TTG from Fig.8(a). Fig.8(a) shows the triconnected component subgraphs $P1$ and $B1$ of the TTG, whereas Fig.8(b) shows the corresponding tree of the triconnected components. The TTG is not normalized: Nodes y and z are incident with multiple incoming and multiple outgoing edges, and all the triconnected component subgraphs of the TTG are fragments. Fig.8(c) shows the normalized version of the TTG from Fig.8(a); it is obtained by splitting nodes y and z, in any order. The normalization introduces edges l and m to the TTG. The tree of the triconnected components of the normalized version consists of four triconnected components: $P1$, $B1$, $P2$, and $B2$ shown in Fig.8(c). It follows from Lemma 1 that they are all fragments.

Fig.9(a) shows the tree of the triconnected components of the normalized version from Fig.8(c). Because of Theorem 1, the tree is the RPST of the normalized version. In Fig.9(b), one can see the RPST without trivial fragments, which correspond to the edges l and m. Note that $P2$ now specifies the same set of edges of the TTG as $B2$. Therefore, we omit $P2$, which is redundant, to obtain the tree given in Fig.9(c).

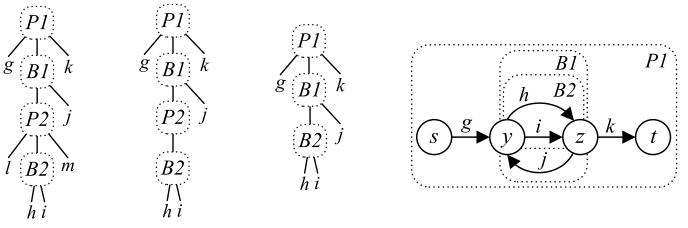


Fig. 9. (a) The tree of the triconnected components of the TTG from Fig. 8(c), (b) the tree from (a) without the fresh edges l and m, (c) the RPST of the TTG from Fig. 8(a), and (d) the TTG from Fig. 8(a) and its canonical fragments

This tree is the RPST of the original TTG from Fig. 8(a). Fig. 9(d) visualizes the TTG again together with its canonical fragments. Please note that Alg. 1, in comparison with the triconnected decomposition shown in Fig. 8(a) and Fig. 8(b), additionally discovered canonical fragment $B2$. $P1$, $B1$, and $B2$ are all the canonical fragments of the TTG.

To show that we indeed obtain the RPST of the original graph, we have to show that (i) each canonical fragment of the normalized version projects onto a canonical fragment of the original graph or onto the empty set, and (ii) for each canonical fragment of the original graph, there is a canonical fragment of the normalized version that is projected onto it. We establish these properties for a single node-splitting step. The claim then follows by induction.

Consider a single node-splitting step transforming a graph G into G', let x be the node that is split into nodes $*x$ and $x*$, and let e be the edge that is added between $*x$ and $x*$. We define the following mappings for the next lemma:

1. A mapping ψ maps a set F of edges of G' to a set $\psi(F)$ of edges of G by $\psi(F) = F \setminus \{e\}$.
2. A mapping ϕ maps a set of edges H of G to a set $\phi(H)$ of edges of G' by $\phi(H) = H \cup \{e\}$ if H has an incoming edge to x as well as an outgoing edge from x, and otherwise $\phi(H) = H$.

Now, we claim:

Lemma 4. *Let ϕ, ψ and e be defined as above. We have:*

1. *If $F \neq \{e\}$ is a fragment of G', then $\psi(F)$ is a fragment of G.*
2. *If H is a fragment of G, then $\phi(H)$ is a fragment of G'.*
3. *If $F \neq \{e\}$ is a canonical fragment of G', then $\psi(F)$ is a canonical fragment of G.*
4. *If H is a canonical fragment of G, then there exists a canonical fragment F of G' such that $\psi(F) = H$.*

The proof of Lemma 4 is in [16]. Lemma 4 and the fact that each step in Alg. 1 can be computed in linear time allow us to conclude:

Theorem 2. *Alg. 1 computes the RPST of a TTG whose completed version is biconnected in linear time.*

4 Generalization of the Refined Process Structure Tree

So far, the RPST decomposition is restricted to TTGs whose completed version is biconnected. In practice this is not sufficient, as a process model may have multiple sources and sinks, cf., Fig.10(b), may be disconnected or may violate biconnectedness assumption. For the latter, consider Fig.10(a). Node u is a separation point of the completed version of the graph as its deletion separates the node labeled with $a1$ from the rest of the graph. Hence, the completed version is not biconnected. Note that process modeling languages such as BPMN and EPC do not impose such structural limitations. In fact, a test of the SAP reference model [17], a collection of industrial process models given as EPCs, showed that more than 80 percent of the models violate one of the restrictions.

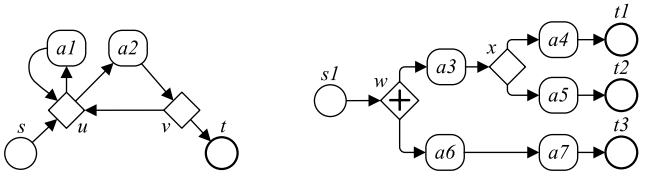


Fig. 10. A workflow graph (a) whose completed version is *not* biconnected, (b) has multiple sinks

In this section, we propose a way to decompose any MTG. The results of this section are also described in detail in a thesis [18]. We start by decomposing arbitrary TTGs.

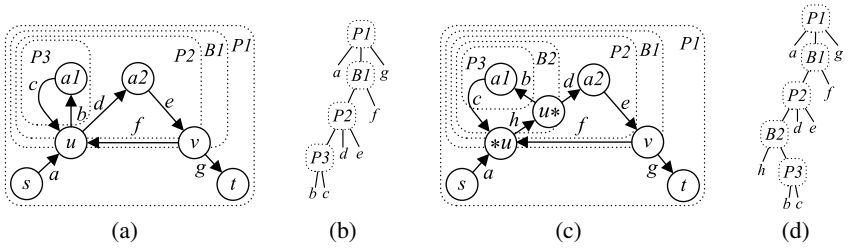
4.1 The RPST of TTGs

Fig.11(a) shows the TTG that corresponds to the process model in Fig.10(a). As we explained above, its completed version is not biconnected because node u is a separation point. Note that u has multiple incoming as well as multiple outgoing edges. Every separation point has this property:

Lemma 5. *Let G be a TTG. Every separation point of $C(G)$ has more than one incoming and more than one outgoing edge in G.*

Proof. A source s and a sink t of G are in the same biconnected component of $C(G)$ as they are connected in G and, therefore, biconnected in $C(G)$ after introducing the return edge. Moreover, it is easy to see that $C(G)$ is connected without s or t and, hence, s and t are not separation points of $C(G)$. Let x, without loss of generality, be some separation point of $C(G)$ that results in a set B of biconnected components. Let $b \in B$, without loss of generality, be a biconnected component induced by x that does not contain s and t. Assume y is a node which belongs to b. As every node of G is on a path from s to t, then x is on every path from s to y and from y to t. A path from s to y implies that x has an incoming edge that does not belong to b and an outgoing edge that belongs to b. A path from y to t implies that x has an incoming edge that belongs to b and an outgoing edge that does not belong to b. Hence, the claim holds.

If b consists of a single edge, it is an incoming and an outgoing edge of x. Every path from s to t through x also contains two edges incident with x, an incoming and an outgoing, which do not belong to b. Hence, the claim holds. □



(a) (b) (c) (d)

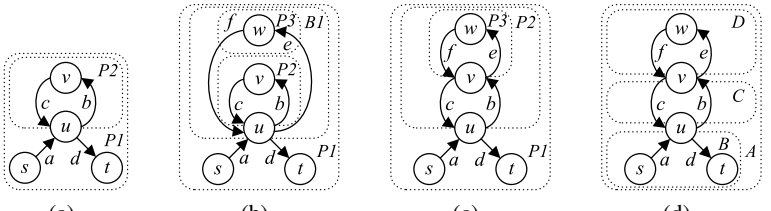
Fig. 11. (a) A TTG whose completed version is *not* biconnected, (b) the RPST of (a), (c) the normalization of (a), and (d) the RPST of (c)

It follows that the completed version of the normalization of G is biconnected. Therefore, we can apply Alg. 1 from Sect. 3.2 to decompose an arbitrary TTG. We call the resulting decomposition of G the *RPST* of G. This is a generalization of the previous definition because if $C(G)$ is already biconnected, we get the RPST as defined previously. Note that we obtain the same result by splitting only the separation points of G, computing the RPST of the resulting graph G' (in any way), and then projecting the RPST of G' onto G. As the normalized version and its RPST are unique, it then follows from the construction that the RPST of an arbitrary TTG is unique.

Fig. 11 shows the RPST of the example, as well as the way in which it is obtained. Again, the RPST of the original graph is obtained by deleting the edge h, which was generated in the node-splitting, and afterwards removing the redundant fragment $B2$.

Figs. 12(a), 12(b), and 12(c) show more examples of decompositions of TTGs whose completed versions are not biconnected. Every subgraph obtained has either exactly two boundary nodes, one entry and one exit, or exactly one boundary node, which is *bidirectional*. Let G be a TTG and F be a connected subgraph of G. A boundary node u of F is *bidirectional* if there exist an incoming and an outgoing edge of u inside F, and there exist an incoming and an outgoing edge of u outside F. Note that control flow can both enter and exit F through u.

Valdes [19] has proposed an alternative way to decompose an arbitrary TTG G. He proposed to first compute the *biconnected components* of $C(G)$ and then further decompose each biconnected component into its triconnected components. If we adapt this idea and compute the RPST of each biconnected component of $C(G)$, we obtain a root component that contains all biconnected components as children, which in turn have their RPSTs as subtrees. The result for the graph from Fig. 12(c) is shown in Fig. 12(d).



(a) (b) (c) (d)

Fig. 12. (a)–(c) The RPST of a TTG, and (d) Valdes's parse tree of the TTG from (c)

which is different from the decomposition we propose. Note that the result has a component that has more than two boundary nodes, e.g., B, and another one having two boundary nodes that are both bidirectional, e.g., C. Unlike our decomposition, the decomposition in Fig. 12(d) does not reflect the fact that the component containing node w depends on the component that is entered through node u.

4.2 The RPST of MTGs

To decompose an arbitrary MTG, we 'normalize' an MTG into a TTG by constructing a unique source and a unique sink as follows.

Definition 3. Let G be an MTG. We construct a graph G' from G as follows.

1. If G has more than one source, a new source s is added and for each source node u of G, an edge from s to u is added.
2. If G has more than one sink, a new sink t is added and for each sink node v of G, an edge from v to t is added.

G' is a TTG, which we call the *TTG version* of G. The *normalized version* G^* of G is the normalized version of G'.

By normalizing an MTG, we again obtain a TTG whose completed version is biconnected. The normalized version can be decomposed with the RPST, and the decomposition can be projected onto the original MTG through Alg. 1. The result that is obtained from applying Alg. 1 to the normalized version of an MTG G is called the *RPST* of G. The RPST of an MTG is unique.

Fig. 13 shows (a) an MTG G, (b) the RPST of G, (c) the TTG version G' of G, and (d) the RPST of G'. The RPST of G is derived from the RPST of G' with Alg. 1.

Note that for an MTG, the subgraphs formed by the decomposition may have more than two boundary nodes. For example, subgraph $B1$ in Fig. 13(a) has two sources u and v as entries, and an exit w. Subgraph $B2$ has an entry w, and three sinks as exits. Subgraph $P1$ two sources as entries, and three sinks as exits.

An RPST-formed subgraph is not necessarily a connected subgraph of an MTG. If an MTG is disconnected, the root fragment of its RPST is a union of the connected components of the MTG. For example, Fig. 14 shows an example of (a) a disconnected MTG G, (b) the RPST of G, (c) the TTG (and normalized) version G^* of G, and (d)

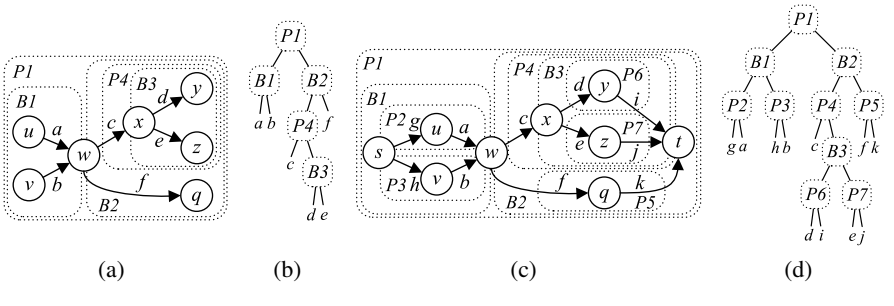


Fig. 13. (a) An MTG G, (b) the RPST of G, (c) the TTG version G' of G, and (d) the RPST of G'

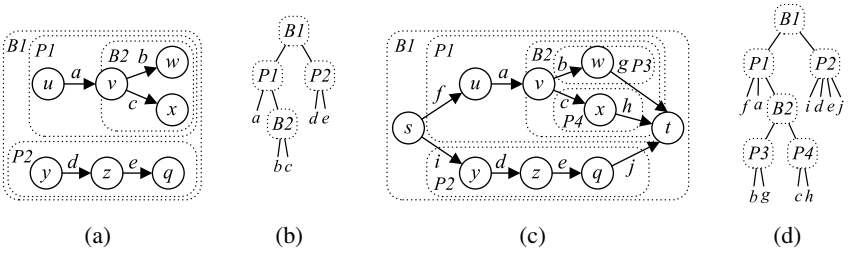


Fig. 14. (a) A disconnected MTG G, (b) the RPST of G, (c) the TTG version G^* of G, and (d) the RPST of G^*

the RPST of G^*. Note that every connected component of the MTG always becomes a separate component of the RPST decomposition.

Fig.15 shows the RPST-formed fragments of the workflow graphs introduced in Fig.10. We can use these fragments to translate BPMN diagrams into BPEL processes. We have labeled the fragments according to the BPEL blocks they correspond to. For example, *sequence B* in Fig.15(a) is a sequence of a *while* loop and the activity *a2*. These decompositions are not directly obtainable with any prior decomposition technique.

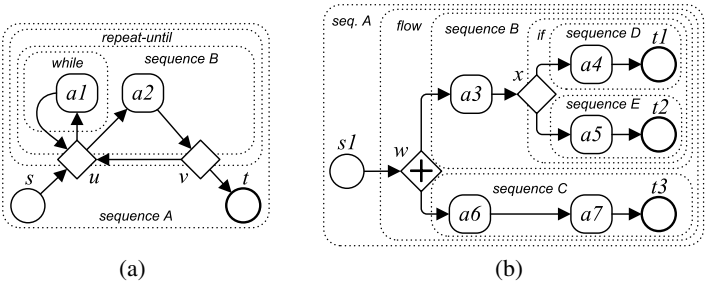


Fig. 15. The RPST-formed fragments of the workflow graphs introduced in Fig.10

5 Conclusion

We simplified the theory for workflow graph parsing into single-entry-single-exit fragments through use of normalized TTGs. This leads to a simplification of the RPST parsing algorithm and its implementation. The implementation effort is essentially reduced to the computation of the triconnected components, of which an implementation is publicly available [14]. In fact, in many applications, nodes have either a single incoming or a single outgoing edge, in which case no pre- and postprocessing steps are required. Together with our previous results [1,18], we have a parsing technique that produces a unique and modular decomposition in linear time in a simple way. The result has a simple characterization in terms of canonical fragments.

In the second part of the paper, we have shown how the RPST technique gives rise to a decomposition of any workflow graph that may occur in practice. The only remaining assumption is that each node must be on a path from some source to some sink.

We have implemented the simplified RPST computation, as proposed in this paper, and tested its functionally against the implementation of the original RPST technique [1] on the SAP reference model [17], which consists of 604 EPC models. The models were transformed to TTGs that range in size from 2 to 195 edges, with the average of 28.7 edges in one TTG. As it was discovered during evaluation, the models have on average 16.5 non-trivial fragments, ranging from the minimum of 1 fragment to the maximum of 132 fragments in one model.

References

1. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data & Knowledge Engineering 68(9), 793–818 (2009)
2. García-Bañuelos, L.: Pattern identification and classification in the translation from BPMN to BPEL. In: Chung, S. (ed.) OTM 2008, Part I. LNCS, vol. 5331, pp. 436–444. Springer, Heidelberg (2008)
3. Polyvyanyy, A., García-Bañuelos, L., Weske, M.: Unveiling hidden unstructured regions in process models. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2009. LNCS, vol. 5870, pp. 340–356. Springer, Heidelberg (2009)
4. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 278–293. Springer, Heidelberg (2009)
5. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In: PLDI, pp. 171–185 (1994)
6. Johnson, R.: Efficient Program Analysis using Dependence Flow Graphs. PhD thesis, Cornell University, Ithaca, NY, USA (1995)
7. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models though SESE decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
8. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and resolving process model differences in the absence of a change log. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
9. Polyvyanyy, A., Smirnov, S., Weske, M.: The triconnected abstraction of process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 229–244. Springer, Heidelberg (2009)
10. Vanhatalo, J., Völzer, H., Leymann, F., Moser, S.: Automatic workflow graph refactoring and completion. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 100–115. Springer, Heidelberg (2008)
11. Battista, G.D., Tamassia, R.: On-line maintenance of triconnected components with SPQR-trees. Algorithmica 15(4), 302–318 (1996)
12. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 4–19. Springer, Heidelberg (2008)
13. Tarjan, R.E., Valdes, J.: Prime subprogram parsing of a program. In: POPL 1980, pp. 95–105. ACM, New York (1980)
14. Gutwenger, C., Mutzel, P.: A linear time implementation of SPQR-trees. In: Marks, J. (ed.) GD 2000. LNCS, vol. 1984, pp. 77–90. Springer, Heidelberg (2001)
15. Hopcroft, J., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM J. Comput. 2(3), 135–158 (1973)

16. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. Technical Report RZ 3745, IBM (2009)
17. Curran, T., Keller, G., Ladd, A.: SAP R/3 Business Blueprint: Understanding the Business Process Reference Model. Prentice-Hall, Inc., Upper Saddle River (1997)
18. Vanhatalo, J.: Process structure trees: Decomposing a business process model into a hierarchy of single-entry-single-exit fragments. PhD thesis, University of Stuttgart, Germany, vol. 1573, dissertation.de — Verlag im Internet (July 2009) ISBN: 978-3-86624-473-3
19. Valdes, J.: Parsing Flowcharts and Series-Parallel Graphs. PhD thesis, Stanford University, CA, USA (1978)

Automated Generation of Web Service Stubs Using LTL Satisfiability Solving

Sylvain Hallé

Université du Québec à Chicoutimi
Chicoutimi (Québec) G7H 2B1, Canada
shalle@acm.org

Abstract. Given a web service W, a *stub* is a simple service S intended to impersonate W and simulate some of its input-output patterns. When W's behaviour is represented by a logic formula φ, S can use a satisfiability solver to drive the simulation and generate valid messages compliant with φ. A satisfiability solver for a variant of first-order temporal logic, called LTL-FO$^+$, is described. Using a chain of existing, off-the-shelf tools, a stub can be generated from a set of LTL-FO$^+$ formulæ expressing a wide range of constraints, including message sequences, parameter values, and inter-dependencies between both. This, in turn, produces a faithful simulation of the original service that can be used for development and testing.

1 Introduction

The distributed nature and loose coupling of web services has contributed to their increasing success by separating various processes into independent units that communicate through standard network protocols. Often a web service under development cannot be run and tested in isolation, and must communicate with its actual partners, even in its early stages of design. Yet for various reasons, it might be desirable that no actual communication takes place.

A possible way to alleviate this issue is to close the environment by replacing an external partner with a *stub*, intended to mimic a web service without providing the actual functionality of the original. A stub typically accepts the same requests and parameters as the actual service, and returns stock responses with the same structure as real ones. The goal is to provide a "dummy" that is sufficiently realistic to be used as a partner for the service under development. However, developing such a stub is itself a time-consuming task. One is required to peruse the documentation of the service to simulate, and write code to parse any incoming request and produce a response with appropriate parameters.

There exist a few development tools that ease this process by automatically producing boilerplate requests and responses, for example by parsing the WSDL document associated with the service; these stock events can then be filled with actual values by the programmer. While such tools allow to produce credible abstractions of simple web services, they fail to capture the transactional nature

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 42–55, 2011.

of some others. An auto-generated stub typically returns the same hard-coded response for every request of the same type. If the original service's behaviour is *stateful*, and includes dependencies on the sequence of requests it receives, one must either do without such aspects in the stub, or resort to code them by hand.

This paper attempts to resolve this issue by describing a method for automatically generating stubs of transactional web services. In addition to the definition of requests, responses and their possible parameters, the generic stub shown in this paper also takes a set of assertions on its sequential behaviour, expressed in an extension of Linear Temporal Logic called LTL-FO$^+$. This logic includes a limited form of quantification over request parameters that allows the expression of data dependencies across requests at different moments in a transaction. A prototype implementation uses a model checker as a background engine to generate a counter-example trace to the *negation* of the original specification; this counter-example is then parsed, and a valid response to any request can be constructed from it. Experimental results show on a simple example how a handful of LTL-FO$^+$ expressions are sufficient to automatically produce a faithful simulation of a transactional web service —including data dependencies— and generate responses in reasonable time.

2 Web Service Stubs

The interaction with most web services is done through the use of the Hypertext Transfer protocol (HTTP) [10]. Under this general setting, two main families of services can be distinguished. The first family uses the Simple Object Access Protocol (SOAP), where an operation and its related parameters are encapsulated into an XML document, which is then sent as a payload inside an HTTP POST request. The structure of each XML document, corresponding to the available operations, is advertised into a WSDL document.

We rather concentrate in this paper on web services using HTTP "GET" requests for their invocation. GET requests invoke only a limited form of payload, consisting of a list of attribute-value pairs. Such parameters are generally appended at the end of a request URL, separated from it by a question mark. For example, a simple GET request for a `login` operation, providing a username (`name`) and an authentication code (`code`), can be encoded as the following URL: `http://example.com/login?user=sylvain&code=12345`. In this case, the web service is hosted on a server named `example.com`.

These services are sometimes called "REST" web services, as they loosely follow a design philosophy called Representational State Transfer [11] that forms the basis of the HTTP protocol. Although REST services accept only a limited form of data structure in their requests, their simplicity became a reason for their wide adoption; it was reported in the early 2000s that 85% of the usage of Amazon's web services API came via the REST interface [20]. The Programmable Web repository lists more than 1,400 APIs using REST [21].

2.1 A Running Example

We devise a simple REST online trading service where a user can create and modify a stock portfolio, adapted from [17]. Table 1 summarizes its available different requests and responses, along with their possible parameters.

Table 1. Summary of operations for the stock portfolio

Operation	Sample request	Sample response
Search for stock	search?term=abcd	stocks?stock=1234&stock=5678...
Get stock info	stock?item=34	stock-info?price=5&name=abc
Create cart	new-cart	cart-info?id=123&empty=true
Add to cart	add?id=123&item=34	cart-info?id=123&empty=false
Remove from cart	remove?id=123&item=34	cart-info?id=123&empty=X
Empty cart	empty?id=123	cart-info?id=123&empty=true
Commit to buy	commit?item=34	confirmed?item=34&item=...
Request payment	req-payment	payment-form
Logout	logout	bye

In this example, the catalogue of available stocks can first be searched using the `search` method, which takes as input a search term and returns a `stocks` response containing a list of available `stock`s. Information on individual items can be asked with the `stock` operation, whose response (`stock-info`) returns its `price` and `name`.

From an inventory of stocks, a portfolio, similar to a shopping cart, can be created. The result of this operation returns a `cart-info`, containing the unique ID used to reference that cart. Elements can be added and removed from the cart, by passing as arguments the cart ID and the item number of the element to add or remove. All such requests are simply replied with `cart-info` as a confirmation.

Finally, a user can benefit from discounted prices if she commits to buy an item. The service replies with a confirmation, listing all the items that were already committed, including the last one. Committed items can no longer be removed from the cart, and any attempt by the user to logout from the system is replied with the payment form instead of the logout confirmation.

Such a simple web service is subject to a number of constraints in the way these operations can be invoked. First, the only allowed parameters for any request and response are those listed in Table 1's respective entries. For example, it does not make sense for a `stocks` response to contain parameter `empty`. Second, all requests must be replied with their appropriate response operation.

However, a correct invocation of this service involves a number of additional constraints linked to the semantics of the operations involved.

1. Since the IDs added to the cart must be valid stock numbers, the service imposes that no element can be added to the portfolio before some stock information has been asked.
2. If a user commits to buy any item during her session, the `logout` message must be replied by a `payment-form` instead of the simple `bye`.

3. Once a cart is created, the same cart `id` must be passed in all requests and responses.
4. Any stock ID sent in a `commit` request must appear in all future `confirmed` messages.

2.2 Producing a Web Service Stub

The REST interface allows this service to be used by any third-party application developer wishing to include stock manipulation functionalities. However, there exist various reasons one might want to avoid communicating with the real service in the development phase.

First, real operations on an actual stock portfolio should be prevented when testing a third-party application. To this end, some providers offer a copy of their actual services running in a closed environment for developers to test with. Amazon Web Services [1] and PayPal [2] provide such "sandboxes". Once an application ends its testing phase, the URL endpoint of the sandbox is replaced by the URL of the real service, which is supposed to work in the exact same way.

While such a principle allows the highest degree of faithfulness, there exist cases where such environments are not available for the service in question. Even when sandboxes do exist, it can be desirable to control the outside environment for debugging purposes. Suppose for example that some piece of local code only executes when the service's responses follow a precise sequence. To debug that piece of code in a sandbox environment, one must find a way for the service to respond that particular sequence —yet the sandboxed service is not under the developer's control.

In such cases, replacing the actual web service with a placeholder *stub* that simulates its input-output patterns can prove an interesting development and testing tool. When hosted locally, the stub can also do away with eventual network problems such as congestion, firewalls and latency.

2.3 Current Solutions

However, as mentioned earlier, stubs mostly require manual programming. We mention a few solutions that have been developed to ease the burden of writing stubs.

A commercial development tool for SOAP web services, called soapUI [3], allows a user to create "mock web services". The WSDL document declaring the structure of each XML request and response is automatically parsed, and a boilerplate message for each is given to the user.

Web services have also been simulated through the generation of random, WSDL-compliant messages when requested. WSDL documents have been used as templates to generate test cases for web services [4]. A tool called TAXI (Testing by Automatically generated XML Instances) has been developed to automatically produce valid instances of XML Schemas [5]. It has been used in the field of web services, to automatically generate XML documents with given structure to be sent as an input to a web service to test. A similar tool

called WebSob [18], when used in conjunction with unit test generators such as JCrasher [8], can generate random WSDL-compliant requests and discover incorrect handling of nonsensical data on the service side.

However, all these approaches treat request-responses as atomic patterns independent of each other, and are ill-suited to the transactional nature of our current example. Formal *grammars* have been used to generate test data [23,19,9,6]. The principle has been extended into *interface grammars*, which have been used to represent the possible sequences of messages in SOAP web services. [16]. The approach takes into account message structure and relationships between message elements [12]. Yet, grammars are monolithic; the same production rule often plays a role in more than one constraint. This makes it hard to take away or to add a new requirement without rewriting substantial parts of the specification. In addition, grammars do not provide an easy way to constrain request parameters.

3 A Formal Model of Web Service Stub Behaviour

To alleviate the issues mentioned above, we first describe a simple formal model of REST web service requests, along with an extension of Linear Temporal Logic suitable to express the dependencies elicited in Section 2.1.

3.1 First-Order Linear Temporal Logic: LTL-FO$^+$

Let $\mathcal{P}$ be the set of *parameter names*, $\mathcal{D}$ be the set of *value names*, and $\mathcal{A}$ be the set of *action names*. We define a special symbol, #, standing for "undefined". A request $r = (a, \star)$ is an element of $\mathcal{A} \times (\mathcal{P} \to 2^{\mathcal{D}} \cup \{\#\})$, where $\to$ designates a function. For example, the request `remove?id=123&item=34` can be represented by the pair $(\text{remove}, \star)$, where $\star$ is the function such that $\star(\text{id}) = \{123\}$, $\star(\text{item}) = \{34\}$, and $\star(p) = \emptyset$ for any other parameter $p \in \mathcal{P}$. A *trace* of requests $\overline{r}$ is a sequence $r_0, r_1, \ldots$ such that $r_i \in \mathcal{R}$ for all $i \geq 0$.

LTL-FO$^+$ is a logic that expresses assertions over traces of requests. Its building blocks are *atomic propositions*, which can be of two forms: "α" for $\alpha \in \mathcal{A}$, and "$p = v$", for $p \in \mathcal{P}$ and $v \in \mathcal{D}$.

A GET request $r = (a, \star)$ satisfies an atomic proposition π, noted $r \models \pi$, exactly when one of the following two cases occurs:

- π is of the form α for $\alpha \in \mathcal{A}$ and $\alpha = a$
- π is of the form $p = v$ for $p \in \mathcal{P}$ and $v \in \mathcal{D}$ and $v \in \star(p)$

Atomic propositions can then be combined with Boolean operators $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not") and $\to$ ("implies"), following their classical meaning. In addition, *temporal operators* can be used. The temporal operator **G** means "globally". For example, the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every request of the trace, starting from the current request. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future request of the trace. The operator **X** means "next"; it is true whenever φ holds in the next request of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all requests until some request satisfies ψ.

Finally, LTL-FO$^+$ adds *quantifiers* that refer to parameter values inside requests. Formally, the expression $\exists_p x : \varphi(x)$ states that in the current request $r = (\alpha, \star)$, there exists a value $v \in \star(p)$ such that $\varphi(v)$ is true. Dually, the expression $\forall_p x : \varphi(x)$ requires that $\varphi(v)$ holds for all $v \in \star(p)$. The semantics of LTL-FO$^+$ are summarized in Table 2, where $\bar{r} = r_0, r_1, \ldots$ is a sequence of requests; the reader is referred to [13] for a deeper coverage of LTL-FO$^+$ in a related context.

Table 2. Semantics of LTL-FO$^+$ for GET requests

$$
\begin{aligned}
\bar{r} \models \alpha &\equiv r_0 = (a, \star) \text{ and } a = \alpha \\
\bar{r} \models p = v &\equiv r_0 = (a, \star) \text{ and } \star(p) = v \\
\bar{r} \models \neg\varphi &\equiv \bar{r} \not\models \varphi \\
\bar{r} \models \varphi \vee \psi &\equiv \bar{r} \models \varphi \text{ and } \bar{r} \models \psi \\
\bar{r} \models \varphi \vee \psi &\equiv \bar{r} \models \varphi \text{ or } \bar{r} \models \psi \\
\bar{r} \models \varphi \rightarrow \psi &\equiv \bar{r} \not\models \varphi \text{ or } \bar{r} \models \psi \\
\bar{r} \models \mathbf{G}\,\varphi &\equiv r_0 \models \varphi \text{ and } \bar{r}^1 \models \mathbf{G}\,\varphi \\
\bar{r} \models \mathbf{F}\,\varphi &\equiv r_0 \models \varphi \text{ or } \bar{r}^1 \models \mathbf{F}\,\varphi \\
\bar{r} \models \mathbf{X}\,\varphi &\equiv \bar{r}_1 \models \varphi \\
\bar{r} \models \varphi\,\mathbf{U}\,\psi &\equiv r_0 \models \psi, \text{ or both } r_0 \models \varphi \text{ and } \bar{r}^1 \models \varphi\,\mathbf{U}\,\psi \\
\bar{r} \models \exists_p x : \varphi &\equiv r_0(p) \neq \# \text{ and } \bar{r} \models \varphi[x/r_0(p)] \\
\bar{r} \models \forall_p x : \varphi &\equiv r_0(p) = \# \text{ or } \bar{r} \models \varphi[x/r_0(p)]
\end{aligned}
$$

3.2 Formalizing Web Service Constraints

Equipped with this language, one can revisit the constraints described in Section 2.1. Constraints on request-responses pairs can be easily expressed; for example, the formula $\mathbf{G}\,(\text{search} \rightarrow \mathbf{X}\,\text{stocks})$ states that every `search` action is followed by a `stocks` action. Similar formulæ can be written for the remaining request-response pairs in Table 1.

The four semantic constraints can also be formalized in LTL-FO$^+$. The following formula express the fact that no element can be added to the portfolio before some stock information has been asked.

$$(\neg\text{add}\,\mathbf{U}\,\text{search}) \vee \mathbf{G}\,\neg\text{add} \tag{1}$$

Similarly, the restriction on commitment to buy can also be written as an LTL-FO$^+$ formula, as follows:

$$\mathbf{G}\,(\text{commit} \rightarrow ((\text{logout} \rightarrow \mathbf{X}\,\text{payment-form})\,\mathbf{U}\,\text{req-payment})) \tag{2}$$

This formula expresses that once a commit request is sent, a `logout` action must be followed by a `payment-form`, unless the form has been requested manually by the client.

The remaining two constraints require the data quantification mechanism specific to LTL-FO$^+$. To express that the same cart ID must be passed in all requests and responses, one writes:

$$\mathbf{G}\,(\exists_{\text{id}} i : \mathbf{G}\,\forall_{\text{id}} j : i = j) \tag{3}$$

This formula states that at any point in a transaction, if the parameter `id` takes some value i, then all occurrences of parameter `id` have a value j such that $i = j$.[1]

Finally, the last LTL-FO$^+$ formula encodes the "memoryful" behaviour of the commitment mechanism:

$$\mathbf{G}\,(\text{commit} \rightarrow \forall_{\text{id}} i : \mathbf{G}\,(\text{confirmed} \rightarrow \exists_{\text{id}} j : i = j)) \tag{4}$$

This formula states that globally, any `id` value i occurring in a `commit` request is such that, from now on, any `confirmed` response will include an occurrence of `id` with value j such that $i = j$. This is equivalent to the requirement that the list of committed items is additive and includes all previously committed items.

3.3 Model Checking and Satisfiability of LTL-FO$^+$

One can see how, by means of four simple LTL-FO$^+$ formulæ (in addition to the straightforward request-response patterns), a somewhat faithful encoding of the service's behaviour can be achieved. That is, any trace of GET requests fulfilling these formulæ must be such that, for example, cart operations will be mimicked with relative precision, the commitment mechanism will faithfully remember all committed items, and will force payment before logout. Consequently, a procedure that can *generate* a sequence of such requests, according to the temporal specifications, can form the basis of a web service stub for the trading service.

To this end, existing tools called *model checkers* can be put to good use. An LTL *model checker* is a piece of software which, given an LTL formula φ and some encoding for a Kripke structure K, can exhaustively check all possible traces of that state machine for compliance to φ. Otherwise, the model checker returns a counter-example: a single execution of the state machine that violates the formula. This counter-example generation mechanism can be put to good use in the present context. To explicitly produce a trace of M that complies with φ, one can send $\neg\varphi$, the negation of the property under study, to the model checker. Any counter-example trace found by the tool is directly a trace satisfying the original formula φ. Hence, a model checker can be used in reverse, as a "model finder".

In the present context, this principle can be used as follows. Given a trace of n requests and responses, a Kripke structure with $n+1$ states is generated. The first n states are completely fixed and correspond exactly to the trace that has been recorded. The values for the last state are left unconstrained. When the model checker finds a counter-example, this scheme forces the counter-example to be identical to the recorded trace for its n first states. Then, by construction, the $n+1$-th state will correspond to a valid extension of the current trace.

4 An Off-the-Shelf Web Service Stub

The use of a model checker as a model finder allows us to use off-the-shelf components to handle most of the computation of a satisfying trace for an

[1] This formula, as well as formula (4), assumes that only one cart per session can be created, and that requests for multiple sessions are split into their own separate trace.

LTL-FO$^+$ specification. In this section, we describe such a tool chain and provide initial experimental results.

4.1 Tool Chain

The process is shown in Figure 1. First, an LTL-FO$^+$ specification is given to the stub (1), which converts it internally into a formula ψ that will be used by the model checker to generate its counter-example. Every time a new request r_i is sent (2), the stub appends it to the trace of previous requests $r_0, \ldots, r_{i-1}$ stored in memory. A linear Kripke structure K is created from that trace, leaving values for a potential state r_{i+1} undefined. This structure, along with formula ψ, is then sent to the model checker (3). If the trace can be extended with a new state in such a way that φ is fulfilled, a counter-example trace for ψ will be found and returned to the stub (4). The output from the model checker is parsed, and the $i+1$-th state of that counter-example constitutes the message to return. An HTTP response is formed from that message and returned by the stub (5).

An interesting consequence of this architecture is that the stub can act either as the server, or in reverse as the *client*. Indeed, one can use the stub to generate requests, which can then be sent to an actual implementation of the web service. The stub generates valid HTTP requests, according to the specification, that are sent to the actual web service. The service responds with actual HTTP responses that are then processed by the stub. This is what Hughes et al. call a *driver* [16]. This is possible because the declarative specification of the service's behaviour gives constraints on responses as well as on requests.

The stub itself is hosted by an Apache web server, and is made of a single PHP script of about 20 kilobytes. This script is responsible for handling the LTL-FO$^+$ specification, keeping in persistent storage the trace of previous requests and

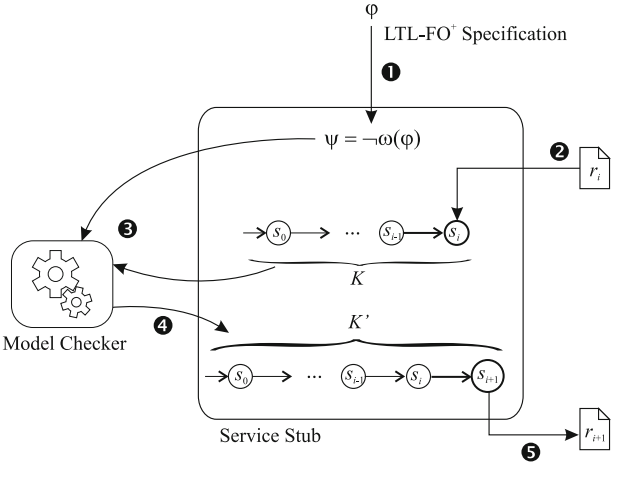


Fig. 1. Tool chain for a web service stub

Table 3. A sample definition file for the web service stub

```
REQUESTS
  new-cart[];
  add[id,item];
  ...
DOMAINS
  id: 0,1;
  item: item1, item2;
  ...
SPECS
  G (add -> cart-info);
  G (<id:i> G [id:j] i=j);
  ...
```

responses, generating appropriate input files for the model checker and parsing its output back into HTTP requests.

The specific model checker used in the chain is NuSMV 2.5 [7]. Although many other proponents could have been considered, recent benchmarks [22] have shown that, when used as LTL satisfiability solvers, symbolic model checkers like NuSMV are clearly superior than explicit model checkers such as SPIN [15]. Since, in the current setting, the whole process is repeated at every web service invocation, both the temporary NuSMV input file and a copy of NuSMV itself are located on a RAM drive and executed from that location.

The PHP script remains the same, regardless of the web service to imitate; only the set of LTL-FO$^+$ formulæ that dictate its behaviour will differ from one stub to the other. The stub requires an input file, whose structure is shown in Table 3. The input file first enumerates the list of possible requests, and gives between brackets the names of all possible parameters attached to that request. It then provides ranges of possible values for each parameter, that are used to populate requests and responses when required. Finally, the input file provides a list of LTL-FO$^+$ formulæ that describe the stub's intended behaviour, as described previously.

4.2 From a Trace to a Kripke Structure

For the purpose of this stub, a straightforward translation of GET requests and LTL-FO$^+$ formulæ was developed. First, the arity of GET requests is fixed to some integer k. Any request can then be represented by a fixed data structure formed of $2k + 1$ variables:

- A variable a, with domain $\mathcal{A}$, containing the request's action
- k variables named p_1 to p_k, with domain $\mathcal{P} \cup \{\#\}$, containing the request's parameter names
- k variables named v_1 to v_k, with domain $\mathcal{D} \cup \{\#\}$ containing the value for each parameter

Intuitively, a GET request simply becomes an action, followed by an array of parameter-value pairs. Since the maximal arity is not necessarily reached in each

request, we allow both p's and v's to contain the "empty" symbol #. A variable p_i equal to # stands for an empty slot in the array.

From a given trace of n GET requests, the stub creates a state machine in NuSMV's input format, whose first n states are completely defined by the contents of that trace. It uses an additional integer, `m_count`, that is incremented at each message. The transition relation of that system, defined by means of the `TRANS` construct, gives a number of conditions on a valid transition:

1. `m_count` is incremented by 1 at each step
2. When `m_count` equals m, variables a, and all the p_i's and v_i's take the values corresponding to request r_m
3. For each i, an empty p_i entails an empty v_i
4. For each $i < j$, an empty p_i entails an empty p_j
5. For each possible action $\alpha \in \mathcal{A}$, $a = \alpha$ constrains the possible parameters that the p_i can take (according to the definition file)
6. For each possible action $p \in \mathcal{P}$, $p_i = p$ constrains the possible values that the v_i can take (according to the definition file)

Conditions 1-2 force the system to have a single execution for its first n states corresponding to the trace of GET requests. Conditions 3-4 force the model checker to produce an $n+1$-th state that corresponds to an actual request: there can't be a value without a parameter, and all empty slots appear at the end of the array. Finally, conditions 5-6 force the model checker to include only valid parameters for a given action, and only valid values for each parameter, as was specified in the `REQUESTS` and `DOMAINS` sections of the stub's definition file.

4.3 From LTL-FO$^+$ to LTL

The use of an off-the-shelf model checker implies that support for LTL-FO$^+$ must be simulated, by converting the original specification back into a classical LTL formula. This translation is performed by a recursive mapping ω, which takes an LTL-FO$^+$ formula and produces an equivalent LTL formula.

The translation is direct for all Boolean and temporal operators; that is, $\omega(\circ\ \varphi) \equiv \circ\ \omega(\varphi)$ for $\circ \in \{\neg, \mathbf{G}, \mathbf{F}, \mathbf{X}\}$, and $\omega(\varphi \circ \psi) \equiv \omega(\varphi) \circ \omega(\psi)$ for $\circ \in \{\vee, \wedge, \rightarrow, \mathbf{U}\}$. The only special case to be handled is quantifiers, which must be expanded into their propositional equivalents:

- $\omega(\exists_p x : \varphi) \equiv \bigvee_{v \in \mathcal{D}} (\bigvee_{i=1}^{k} p_i = p \wedge v_i = v \wedge \omega(\varphi[x/v]))$
- $\omega(\forall_p x : \varphi) \equiv \bigwedge_{v \in \mathcal{D}} (\bigwedge_{i=1}^{k} (p_i = p \wedge v_i = v) \rightarrow \omega(\varphi[x/v]))$

Intuitively, the translation of $\exists_p x : \varphi$ indicates that for some value $v \in V$, one of the p_i contains parameter p and its corresponding v_i contains some value $v \in V$; moreover, this value is such that $\varphi[x/v]$ is true. Since parameter p can occur in any slot of the request, the formula must be repeated for every p_i. Similarly, the translation of $\forall_p x : \varphi$ stipulates that whenever some value v is contained in a state variable v_i such that the corresponding p_i has value p, then $\varphi[x/v]$ is true. The length of the resulting LTL expression is exponential in the number k of quantifiers, with respect to the original LTL-FO$^+$ formula.

4.4 Experimental Results

To prove the concept, we performed a series of experiments with the stock portfolio stub. We first measured the running time at each invocation of the stub for a given interaction. That is, we started the stub from an empty trace, and repeatedly invoked it to generate a possible first, second, third message, and so on. The results are plotted in Figure 2.

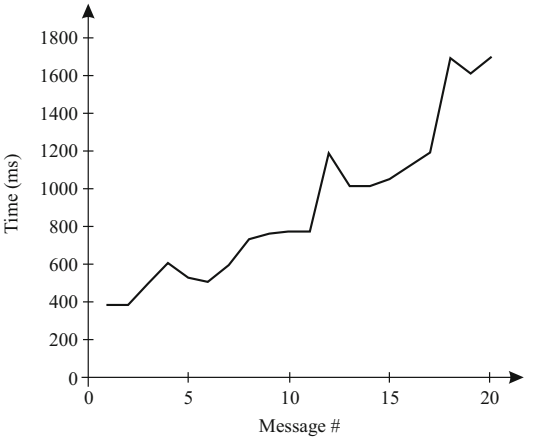


Fig. 2. Stub running time per message for a trace of 20 requests and responses

Since the model checker is given a larger Kripke structure each time (it contains one more state), correspondingly, we expect the running times to generate each new message to increase as the transaction progresses. As the figure shows, running times for a transaction range between 0.4 and 1.7 second and steadily increase as the trace gets longer. However, these times can be deemed reasonable, in a context where a stub is intended for use as a placeholder when testing a third-party client.

Finally, since LTL-FO$^+$ involves quantification over data domains, we measured the impact of the domain size on generation time. More precisely, we generated a fixed trace of 20 requests, and called the stub to extend that trace with a compliant 21st request, and measured its running time. At first, the stub was given a specification where the `item` parameters could take a single value, and repeated the process by progressively increasing the domain size, up to 12 possible item numbers. The results are shown in Figure 3.

As one can see, domain size has the highest impact on the stub's running time. Despite a drop in execution time when the domain reaches a size of 6,[2] the global tendency is an exponential growth. For the properties of the stock portfolio example, as soon as the data domain reaches 3 to 4 elements, running times become

[2] This drop is most likely due to NuSMV's opportunity to optimize its internal representation of the system once it reaches a certain number of states. The exact cause was not investigated, and is out of the scope of this paper.

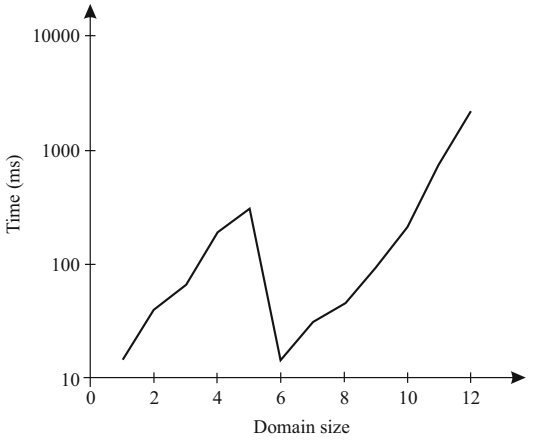


Fig. 3. Stub running time per message with varying data domain sizes

prohibitive. This issue can be mitigated by the fact that, again, the stub is intended to be used as a tool for testing a third-party client; as a result, a domain of only a few elements is probably sufficient to cover most possible behaviours.

5 Conclusion and Open Issues

The automated generation of web service stubs based on temporal logic specifications appears to be a promising path. The case study described in this paper shows how a faithful description of a service behaviour can be captured by only a few expressions in an extension of Linear Temporal Logic called LTL-FO$^+$. These expressions can even stipulate data dependencies spread across requests that can be arbitrarily far apart in an execution; therefore, they are especially suited to describe *transactional* web service behaviour.

In turn, preliminary experimental results indicate that the use of a model checker as a back-end engine to produce message traces compliant with these formulæ is feasible. Despite a potential exponential growth in execution time with respect to domain size, for modest domain sizes, it achieves response times appropriate for its use as a development and testing tool for third-party applications. It shall also be recalled that current solutions, such as mock web services with hard-coded stock responses, do not provide any variability in requests' and responses' values, and can be likened to the current stub with a domain of size one. In this respect, even small domains can be seen as an interesting new feature.

An extended version of the stub mechanism described in this paper is currently under development. The concepts presented hereby lend themselves to possible improvements and raise a number of open issues. We conclude this paper by briefly mentioning a few of them.

1) Messages are considered flat. Although this simplification is appropriate for REST web services, where all arguments of a call are linearized into a list of

parameter-value pairs, there exist cases where nested message structures, such as XML messages using the SOAP protocol, are necessary. Taking into account so-called *semi-structured* requests, and modelling them appropriately for input into a model checker is a challenging problem.

2) Explicit handling of quantification. The use of NuSMV as a back-end engine requires that LTL-FO$^+$ expressions be translated into LTL. Currently, the quantification is removed in an explicit way, i.e. each quantifier is replaced with the appropriate conjunction or disjunction over the range of possible values. An alternate translation of quantifiers into classical temporal logics, called *freeze quantification*, has been shown to improve model checking performance in such situations [14].

3) Reliance on model checking. The main argument in favour of using a model checker as the satisfiability engine is the presumption that its exhaustive state space search is more efficient than any home-brewed constraint solver specific to LTL-FO$^+$. However, retrofitting an existing on-the-fly model checking algorithm for LTL-FO$^+$ [13] into a constraint solver might prove equally successful.

4) Incremental support. As was described earlier, the whole chain of generating a Kripke structure from the most recent trace of messages, starting NuSMV to find a counter-example, and process the returned trace back into a message has to be repeated every time the stub is invoked. Yet most of the work done by the model checker on a trace of n messages could be reused for the trace of $n + 1$ sharing a prefix of length n with the previous round. Support for an "incremental" evaluation of the trace, where information computed by previous rounds of solving could be saved and resumed, could greatly improve performance of the stub. Based on the previous figures, this could amount to an average running time of 70 ms per message.

References

1. Amazon flexible payments service sandbox, `http://docs.amazonwebservices.com/AmazonFPS/latest/SandboxLanding/index.html` (retrieved May 28, 2010)
2. PayPal sandbox user guide. Technical Report 100007.en_US-200910 (October 2009), `https://cms.paypal.com/cms_content/US/en_US/files/developer/PP_Sandbox_UserGuide.pdf` (retrieved May 28, 2010)
3. soapUI: the web services testing tool (2009), `http://www.soapui.org/`
4. Bai, X., Dong, W., Tsai, W.-T., Chen, Y.: WSDL-based automatic test case generation for web services testing. In: IEEE International Workshop on Service-Oriented System Engineering (SOSE) 2005, pp. 207–212 (2005)
5. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: Towards automated WSDL-based testing of web services. In: Bouguettaya, A., Krüger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 524–529. Springer, Heidelberg (2008)
6. Bauer, J.A., Finger, A.B.: Test plan generation using formal grammars. In: Proceedings of the 4th International Conference on Software Engineering, Munich, Germany, pp. 425–432 (September 1979)

7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
8. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Exper. 34(11), 1025–1050 (2004)
9. Duncan, A.G., Hutchison, J.S.: Using attributed grammars to test designs and implementations. In: Proceedings of the 5th International Conference on Software Engineering, New York, NY, USA, pp. 170–178 (March 1981)
10. Fielding, R.J., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – HTTP/1.1. Technical Report RFC 2616, IETF (June 1999)
11. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
12. Hallé, S., Hughes, G., Bultan, T., Alkhalaf, M.: Generating interface grammars from WSDL for automated verification of web services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 516–530. Springer, Heidelberg (2009)
13. Hallé, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: EDOC, pp. 63–72. IEEE Computer Society, Los Alamitos (2008)
14. Hallé, S., Villemaire, R., Cherkaoui, O.: Specifying and validating data-aware temporal web service properties. IEEE Trans. Software Eng. 35(5), 669–683 (2009)
15. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, Reading (2003)
16. Hughes, G., Bultan, T., Alkhalaf, M.: Client and server verification for web services using interface grammars. In: Bultan, T., Xie, T. (eds.) TAV-WEB, pp. 40–46. ACM, New York (2008)
17. Josephraj, J.: Web services choreography in practice (2005), `http://www-128.ibm.com/developerworks/webservices/library/ws-choreography/`
18. Martin, E., Basu, S., Xie, T.: Automated testing and response analysis of web services. In: ICWS, pp. 647–654. IEEE Computer Society, Los Alamitos (2007)
19. Maurer, P.M.: Generating test data with enhanced context-free grammars. IEEE Software 7(4), 50–55 (1990)
20. O'Reilly, T.: REST vs. SOAP at Amazon (2003), `http://www.oreillynet.com/pub/wlg/3005` (retrieved June 2, 2010)
21. Programmable Web. Web services directory (2010), `http://www.programmableweb.com` (retrieved June 1, 2010)
22. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
23. Sirer, E., Bershad, B.N.: Using production grammars in software testing. In: Proceedings of DSL 1999: the 2nd Conference on Domain-Specific Languages, Austin, TX, US, pp. 1–13 (1999)

Passive Testing of Web Services*

César Andrés[1], M. Emilia Cambronero[2], and Manuel Núñez[1]

[1] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
[2] Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha, Spain
c.andres@fdi.ucm.es, emicp@dsi.uclm.es, mn@sip.ucm.es

Abstract. This paper presents a methodology to perform passive testing based on invariants of distributed systems with time information. This approach is supported by the following idea: A set of invariants represents the most relevant expected properties of the implementation under test. Intuitively, an invariant expresses the fact that each time the system under test performs a given sequence of actions, then it must exhibit a behavior reflected in the invariant. We call these invariants *local* because they only check the correctness of the logs that have been recorded in each isolated system.

We discuss the type of errors that are undetectable by using only local invariants. In order to cope with these limitations, this paper introduces a new family of invariants, called *globals* to deal with more subtle characteristics. They express properties of a set of systems, by making relations between the set of recorded local logs. In addition, we show that global invariants are able to detect the class of undetected errors for local invariants.

Keywords: Passive Testing, Service Oriented Systems, Monitoring.

1 Introduction

Testing consists in checking the conformance of a system by performing experiments on it. The application of formal testing techniques [11] to check the correctness of a system requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behavior. In this line, the time consumed by each operation should be considered critical in a real-time system. The testing community has shown a growing interest in extending these frameworks so that not only functional properties but also quantitative ones could be tested.

Most formal testing approaches consist in the generation of a set of tests that are applied to the implementation in order to check its correctness with respect to a specification. Thus, testing is based on the ability of a tester to stimulate the

* Research partially supported by the Spanish MEC project TESIS TIN2009-14312-C02.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 56–70, 2011.

implementation under test (IUT) and check the correction of the answers provided by the implementation. However, in some situations this activity becomes difficult and even impossible to perform. Actually, it is very frequent that the tester is unable to interact with the implementation under test. In particular, such interaction can be difficult in the case of large systems working 24/7 since this interaction might produce a wrong behavior of the system. Thus, usually we can classify testing methodologies into two approaches: *Active* and *passive* testing. The main difference between them is how a tester can interact with the IUT. In the active paradigm the tester is allowed to apply any set of tests to the IUT. In passive testing, the tester is only an observer of the IUT, and he has to provide a degree of confidence of the system, only taking into account the monitored traces.

In this paper we present a formal passive testing methodology to test web services with temporal restrictions. In our passive testing paradigm, testers provide a set of *invariants* that represents the most relevant properties that they would like to check against the logs. Our approach makes use of the ideas presented in [3], a timed extension of the framework defined in [6], to define invariants, in [9] to define orchestrator and choreography behaviors, and in [2], our previous approach to test web services using invariants without considering a timed environment. In this paper, a Service-Oriented Scenario consists of a large number of services that interact with each other and testers are unable to interact with them. Thus, we propose passive testing techniques based on invariants to test these systems. These invariants allow us to express temporal properties that must be fulfilled by the system. For example, we can express that the time the system takes to perform a transition always belongs to a specific interval. Thus, timed invariants are used to express the temporal restrictions of the collected logs of the systems. When these events are collected into each service they are called local logs and the invariants to check their correctness are called local invariants.

Our invariants can be seen as the SLAs presented in [12], but there are some relevant differences. In [12] the set of SLAs are formulas that are extracted from the most frequent patterns of the specification. So, these formulas do not contradict the specification. In our approach, we assume that invariants are provided either by a tester or by using data mining techniques [4]. So, the correctness of the invariants must be checked with respect to the specification. Most importantly we do not need to exchange additional information between web services since we have a decentralized approach.

There already exist several approaches to study the integration of formal testing in web services, providing formal machinery for representing and analyzing the behavior of communicating concurrent/distributed systems. Next we briefly review some previous work related to our framework. In [8] an automatic backbox testing approach for WS-BPEL orchestrations was presented, which was based on translation into Symbolic Transition System and Symbolic Execution. This approach supports the rich XML-based data types used in web services without suffering from state explosion problems. This work was inspired in a

previous work on symbolic testing [10], where the authors showed a modeling and testing approach focused on detecting failures, supporting conformance, and reducing drastically the effort to design test cases, validate test coverage, and execute test cases. In [5] a methodology to automatically generate test cases was presented. The authors combine coverage of web services operations with data-driven test case generation. These test cases were derived from WSDL [13] descriptions. For that purpose, they used two tools: soapUI and TAXI. The first one generates stubs of SOAP calls for the operations declared in a WSDL file. The other one facilitates the automated generation of XML instances from an XML Schema. In [7] the authors present different mechanisms to collect traces. They also study the differences between online and offline monitoring, being the main difference that the testing algorithms in online monitoring are adapted to analyze the information as soon as possible, so a huge amount of computational resources is needed. Their methodology differs from our approach since they use the specification to check the correctness of the traces, while we might have only invariants, and they record *global* traces while we can operate at a *local* level.

Regarding our methodology, our invariants can be seen as the SLAs presented in [12] but there are some relevant differences. In [12] the set of SLAs are formulas that are extracted from the most frequent patterns of the specification. So, these formulas do not contradict the specification. In our approach, we assume that invariants are provided either by a tester or by using data mining techniques.

The rest of this paper is structured as follows. First, Section 2 presents our formal framework to represent web services choreographies, and orchestrations. In Section 3 we present how to define local and global invariants. Finally, in Section 4 we present our conclusions and some lines for future work.

2 Preliminaries

In this section we present our formalism to define web services and choreography models. We follow the ideas underlying the definition of orchestration and choreography model behaviors presented in [9]. However, instead of Finite State Machines, we use *Timed Automata*, with a finite set of clocks over a dense time domain, to represent orchestrations. Since we will not use most of the technical machinery behind Timed Automata, the reader is referred to [1]. The internal behavior of a *web service* is given by a Timed Automaton where the clock domain is defined in $\mathbb{R}_+$. The choice of a next state in the automaton does not only depend on the action, but also on the timed constraints associated to each transition. Only when the time condition is satisfied by the current values of the clocks, the transition can be triggered. We assume that the communication between systems is asymmetric.

Definition 1. A *clock* is a variable c in $\mathbb{R}_+$. A set of clocks will be denoted by $\mathcal{C}$. A *timed constraint* φ on $\mathcal{C}$ is defined by the following EBNF:

$$\varphi ::= \varphi \wedge \varphi \mid c \leq t \mid c < t \mid \neg\varphi$$

where $c \in \mathcal{C}$ and $t \in \mathbb{R}_+$. The set of all timed constraints over a set $\mathcal{C}$ of clocks is denoted by $\phi(\mathcal{C})$.

A *clock valuation* ν for a set $\mathcal{C}$ of clocks assigns a real value to each of them. For $t \in \mathbb{R}_+$, the expression $\nu + t$ denotes the clock valuation which maps every clock $c \in \mathcal{C}$ to the value $\nu(c) + t$. For a set of clocks $\mathcal{Y} \subseteq \mathcal{C}$, the expression $\nu[\mathcal{Y} := 0]$ denotes the clock valuation for $\mathcal{C}$ which assigns 0 to each $c \in \mathcal{Y}$ and agrees with ν over the rest of the clocks. The set of all clock valuations is denoted by $\Omega(\mathcal{C})$.

Let ν be a clock valuation and φ be a timed constraint. We write $\varphi \vdash \nu$ iff ν holds φ; $\varphi \nvdash \nu$ denotes that ν does not hold φ. □

Next we define a *web service*. The internal behavior of a web service, in terms of its interaction with other web services, is represented by a *Timed Automaton*.

Definition 2. We call any value $t \in \mathbb{R}_+$ a *fixed time value*. For all $t \in \mathbb{R}_+$ we have both $t < \infty$ and $t + \infty = \infty$. We say that $\hat{p} = [p_1, p_2]$ is a *time interval* if $p_1 \in \mathbb{R}_+$, $p_2 \in \mathbb{R}_+ \cup \{\infty\}$, and $p_1 \leq p_2$. We consider that $\mathcal{I}_{\mathbb{R}_+}$ denotes the set of time intervals.

Along this paper ID denotes the set of web service identifiers. A *web service* is a tuple $\mathcal{A} = (id, \mathcal{S}, s_0, \Sigma, \mathcal{C}, \mathcal{Z}, \mathcal{E})$ where $id \in ID$ is the identifier of the service, $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state, Σ is the alphabet of actions, $\mathcal{C}$ is a finite set of clocks, $\mathcal{Z} : \mathcal{S} \to \phi(\mathcal{C})$ associates a time condition to each state, and $\mathcal{E} \subseteq \mathcal{S} \times \{id\} \times \Sigma \times ID \times \phi(\mathcal{C}) \times \wp(\mathcal{C}) \times \mathcal{S}$ is the set of transitions. We will consider that Σ is partitioned into two (disjoint) sets of *inputs* denoted by $\mathcal{I}$, preceded by ?, and *outputs* denoted by $\mathcal{O}$, preceded by !. Along this paper Σ^{ID} denotes the set $ID \times \Sigma \times ID$.

We overload the $\vdash$ symbol. Let $s \in \mathcal{S}$ and $\nu \in \Omega(\mathcal{C})$. We denote by $s \vdash \nu$ the fact that ν holds $\mathcal{Z}(s)$ (resp. $s \nvdash \nu$ represents that ν does not hold $\mathcal{Z}(s)$). Let $e = (s, id, \alpha, id', \varphi, \mathcal{Y}, s') \in \mathcal{E}$. We denote by $e \vdash \nu$ the fact that ν holds φ (resp. $e \nvdash \nu$ represents that ν does not hold φ). □

Intuitively, a transition $(s, id, \alpha, id', \varphi, \mathcal{Y}, s')$ indicates that if the system is at state s and the current valuation of the clocks holds φ, then the system moves to the state s' performing the action α from id to id' and resetting the clocks in $\mathcal{Y}$. In other words if we consider $e_1 = (s, id, ?\alpha, id_b, \varphi, \mathcal{Y}, s')$ then the action α is emitted from id to id_b, and if we consider $e_2 = (s, id, !\alpha, id_b, \varphi, \mathcal{Y}, s')$ then the action α is received on id from id_b. For each state s, $\mathcal{Z}(s)$ represents a timed constraint for s, that is, the system can remain in s while the current valuation of the clocks holds $\mathcal{Z}(s)$. We will assume the following usual condition on timed automata: For all $s \in \mathcal{S}$ and all valuation $\nu \in \Omega(\mathcal{C})$ if $s \nvdash \nu$ then there exists at least a transition $e = (s, id, \alpha, id', \varphi, \mathcal{Y}, s') \in \mathcal{E}$ with $e \vdash \nu$. This property allows to leave a state once the restrictions on clocks do not hold in that state.

Example 1. Next we present a small running example to explain the previous concepts. Let us consider the set of four web services represented in Figure 1. In this example we will consider that we have only one clock for each web service, and it is set to 0 in the transition that reach the state s_1. The time constrains

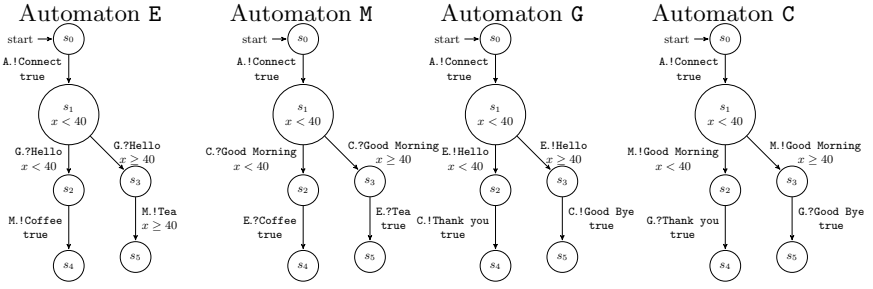


Fig. 1. Example of four web services

represented in some states, for example $x < 40$ in s_1 of `G`, represent the time that this automaton is allowed to be in the state. In the transitions are represented two items. The first one is a pair composed of the id of the web service that interacts with this automaton and the action that they exchange. The second one represents the condition to trigger this transition.

These automata are communicating each others. All of them start receiving `Connect` from another web service, called `A`, in order to be synchronized. After this, each one has its own behaviour. Let us remark that the time between receiving `Connect`, and to perform next action will decide the future behaviour of the web service. For example if we consider that this time is less than 40 time unit and `M` sends `Good Morning` to `C`. Then after any elapsed of time it only can send to `E` the message `Tea`. □

The *semantics* of a web service is given by translating it into a *labeled transition system* with an uncountably number of states. Let us remark that, in general, we will not construct the associated labeled transition system; we will use it to reason about the traces of the corresponding timed automaton.

Definition 3. A *Labeled Transition System*, in short `LTS`, is defined by a tuple $\mathcal{M} = (ID, \mathcal{Q}, q_0, \Sigma, \rightarrow)$, where ID is a set of web service identifiers, $\mathcal{Q}$ is a set of states, $q_0 \in \mathcal{Q}$ is the initial state, Σ is the alphabet of actions, and the relation $\rightarrow \subseteq \mathcal{Q} \times \Sigma^{ID} \cup \mathbb{R}_+ \times \mathcal{Q}$ represents the set of transitions.

Let $\mathcal{A} = (id, \mathcal{S}, s_0, \Sigma, \mathcal{C}, \mathcal{Z}, \mathcal{E})$ be a web service. Its *semantics* is defined by its associated `LTS`, $\mathcal{A}_{\mathcal{M}} = (ID, \mathcal{Q}, q_0, \Sigma, \rightarrow)$, where $\mathcal{Q} = \{(s, \nu) \mid s \in \mathcal{S} \wedge \nu \in \Omega(\mathcal{C}) \wedge s \vdash \nu\}$, $q_0 = (s_0, \nu_0)$, being $\nu_0(c) = 0$ for all $c \in \mathcal{C}$, and we apply two rules in order to generate the elements of $\rightarrow$. For all $(s, \nu) \in \mathcal{Q}$ we have:

- If for all $0 \leq t' \leq t$ we have $s \vdash (\nu + t')$, then $((s, \nu), t, (s, \nu + t)) \in \rightarrow$.
- If $e \vdash \nu$, for $e = (s, id, \alpha, id', \varphi, \mathcal{Y}, s') \in \mathcal{E}$, then $((s, \nu), (id, \alpha, id'), (s', \nu[\mathcal{Y} := 0])) \in \rightarrow$.

In addition, we consider the following conditions: (a) If we have $q \xrightarrow{t} q'$ and $q' \xrightarrow{t'} q''$, then we also have $q \xrightarrow{t+t'} q''$ and (b) if $q \xrightarrow{0} q'$ then $q = q'$, that is, a passage of 0 time units does not change the state. The set of all `LTS` associated with web services will be denoted by `SetLTS`. □

Next, we introduce the notion of *visible trace*, or simply *trace*. As usual, a trace is a sequence of visible actions and time values.

Definition 4. Let $\mathcal{M} = (ID, \mathcal{Q}, q_0, \Sigma, \rightarrow)$ be a LTS, $\vartheta_1, \ldots, \vartheta_n \in \Sigma^{ID}$, and $t_1, \ldots, t_{n-1} \in \mathbb{R}_+$. We say that $\sigma = \langle \vartheta_1, t_1, \vartheta_2, t_2, \ldots, t_{n-1}, \vartheta_n \rangle$, with $n > 0$, is a visible trace, or simply trace, of $\mathcal{M}$ if there exits the transitions (q_1, ϑ_1, q_2), $(q_2, t_1, q_3), \ldots, (q_{2*n-2}, t_{n-1}, q_{2*n-1}), (q_{2*n-1}, \vartheta_n, q_{2*n}) \in \rightarrow$. We will denote by $\mathtt{NT}(\mathcal{M})$ the set of all visible traces.

We define the function TT as the sum of all time values of a normalized visible trace, that is $\mathtt{TT}(\sigma) = \sum_{i=1}^{n-1} t_i$. We denote by $\sigma_{<<} \subseteq \mathtt{SetNVT}$ the set of all subsequences of σ that are visible traces. □

We will usually consider normalized visible traces since this is what we observe from the execution of a system. We cannot observe either internal activity (that is, the performance of internal actions) or different passages of time associated to different transitions. Logs recorded from a IUT, will look like visible traces.

Example 2. Let us consider the web services presented in Figure 1, and $\mathtt{M.LOG} = \langle \mathtt{A.!Connect}, 60, \mathtt{C.?Good\ Morning}, 50, \mathtt{E.?Tea}, \rangle$ be a local log recorded in the web service M. Intuitively, this log represents that A sends Connect to M in order to synchronize with the others web services. Then, after 60 time units, M has sent to C the message Good Morning, and after 50 local time units, it has sent to E the action Tea. □

Next we introduce our formalism to represent *choreographies*. Contrarily to systems of orchestrations, choreographies focus on representing the interaction of web services as a whole. Thus a single machine, instead of the composition of several machines, is considered. The choreography model also is a timed automata, but there are the following differences with respect to web services model: The first one is that there exists only one clock, that is called global clock; the second one is that in the transitions is represented the interaction of the web services, and the third one is that the valuation of the global clock in the initial state is 0, and it can not be reseted.

Definition 5. A *choreography machine* is a tuple $\mathcal{D} = (\mathcal{S}, \Sigma, ID, s_0, \{x\}, \mathcal{Z}, \mathcal{T})$ where $\mathcal{S}$ denotes the set of states, Σ is the set of messages, ID is the set of web service identifiers, $s_0 \in \mathcal{S}$ is the initial state, $\{x\}$ is a clock called global clock, $\mathcal{Z} : \mathcal{S} \rightarrow \phi(\{x\})$ associates a time condition to each state, and $\mathcal{T} \subseteq \mathcal{S} \times \Sigma^{ID} \times \mathcal{S}$ is the set of transitions. The initial valuation of $\{x\}$ is 0. And this clock can never be reseted. □

The notions of traces, and the transformation of the choreography into its LTS associated are similar that the one presented for the web services. Concerning choreography machines, transitions are tuples $(s, id, \alpha, id', \varphi, s')$ where $s, s' \in \mathcal{S}$ are the initial and final states, α is the message, and $id, id' \in ID$ are the sender and the addressee of the message, respectively. Next, let us introduce the idea of choreography with the following example.

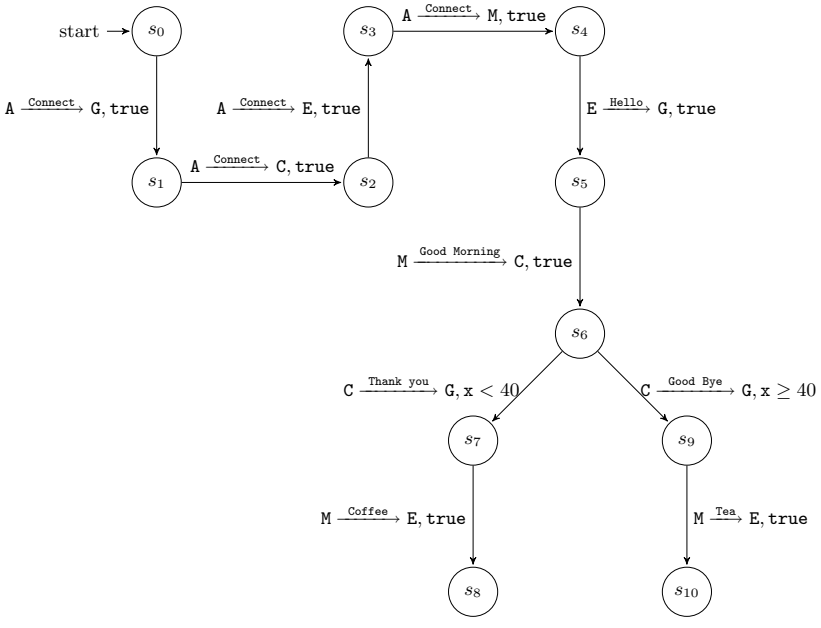


Fig. 2. Choreography of C, E, G and M

Example 3. Let us consider the choreography represented in Figure 2. In this model is denoted the global behaviour of the web services presented in Figure 1. Let us denote that the initial valuation of the global clock, by means x is 0.

In the figure, the transition $s_4\texttt{E} \xrightarrow{\texttt{Hello}} \texttt{G}, \texttt{true}, s_5$ represents that the web service E will send the message Hello to G. The value true means that there not exists any time constrain associated with this transition. This choreography is a reduced graph of the complete choreography of these web services. In the complete one we will have to increase with the permutation of all possible interaction of the web services from s_0 to s_6, from s_6 to s_8 and from s_6 to s_{10}. □

3 Invariants

In this section we introduce the notion of *invariant*. Invariants are used in our approach to represent the properties that we would like to check against the logs extracted from the IUT. The notion of invariant being *correct* with respect to a specification means that if the invariant detects a mismatch, then the implementation that has generated this log is incorrect with respect to the specification. First, after producing a set of invariants and before checking them against the log, they must be checked against the specification; otherwise, we might have an invariant which indicates an erroneous behavior that does not violate the requirements expressed in the specification. Another possibility would be to consider that invariants are correct *by definition*. In this case a mismatch will automatically imply that a fault was detected.

We present two different kinds of invariants: *Local invariants* and *global invariants*. The first type is used to express properties of isolated web services, while the second one, will use the combination of all isolated logs, to check some significant properties at the system level.

3.1 Local Invariants

Definition 6. Let $\mathcal{A} = (id_a, \mathcal{S}, s_0, \Sigma, \mathcal{C}, \mathcal{Z}, \mathcal{E})$ be a web service. We say that the sequence ϕ_{id_a} is a *local invariant* for $\mathcal{A}$ if it is defined according to the following EBNF:

$$\begin{aligned} \phi_{id_a} &::= \texttt{Body} \mapsto \texttt{Consequent} \\ \texttt{Body} &::= \vartheta/\hat{p}, \texttt{Body} \mid \star/\hat{p}, \texttt{Body} \mid \vartheta'/\hat{p} \\ \texttt{Consequent} &::= O \rhd \hat{p} \end{aligned}$$

In this expression we consider that $\hat{p} \in \mathcal{I}_{\mathbb{R}_+}$, $\vartheta' \in \{id_a\} \times \Sigma \times ID$, $\vartheta \in \{id_a\} \times \Sigma \cup \{?\} \times ID$, and $O \subseteq \{id_a\} \times \Sigma \times ID$.

Let $\phi_{id} = \mathcal{P} \mapsto \mathcal{R}$ be a local invariant. We define the functions $\texttt{Body}(\phi_{id}) = \mathcal{P}$ and $\texttt{Consequent}(\phi_{id}) = \mathcal{R}$. The set of all invariants for a set of web services identifiers ID is denoted by Φ_{ID}, and the set of all bodies of these invariants is denoted by Φ_{ID}^{body}. □

Let us remark that time conditions established in invariants are given by intervals. However, web services in our formalism present fix time. Intervals represent the idea that it can be admissible that the execution of a task sometimes takes more time than expected: If most of the times the task is performed on time, a small number of delays can be tolerated. Moreover, another reason for the tester to allow imprecisions is that the artifacts measuring time while testing a system might not be as precise as desirable. In this case, an apparent wrong behavior due to bad timing can be in fact correct since it may happen that the *clocks* are not working properly.

In our framework, the symbol ? can replace any action while the symbol $\star$ can replace a sequence of actions not containing the first action symbol that appears in the part of the invariant that follows it. Intuitively, the EBNF expresses that a local invariant is either a sequence of symbols where each component, but the last one, is either or an expression $id_a, \alpha, id_b/\hat{p}$, with id_a and id_b being web services identifier, with α being an action or the wildcard character ?, and $\hat{p}$ being a timed interval, or an expression $\star/\hat{p}$.

There are two restrictions to this rule: a local invariant cannot contain two consecutive components $\star/[p_1, p_2]$ and $\star/[q_1, q_2]$ since this situation could be simulated by means of the expression $\star/[p_1 + q_1, p_2 + q_2]$, and a local invariant cannot present a component of the form $\star/\hat{p}$ followed by a wildcard character ?, that is, the action of the next component must belong to Σ. The last component, corresponding to the expression $\vartheta'/\hat{p} \mapsto O \rhd \hat{q}$, is composed of two web service identifiers associated with an action, that is ϑ', followed by a timed interval, and followed by a set of triples identifier/actions/identifier and another time interval.

When we check a log with respect to a local invariant, first we check if the log *matches* the body of the invariant. When we find a sequence that matches,

then we check the correctness of this sequence. The correctness of a sequence can have the usual three valued valuations: *correct*, *incorrect* and *inconclusive*. The result is returned inconclusive if the trace never matches the body of the invariant. An invariant can detect an error with respect to two restrictions. These are represented in the consequent part of the invariant, that is $O \rhd \hat{q}$. The first requirement is given by O, and it is associated to the last term of the log. It means that if a log $\langle \vartheta_1, \ldots, \vartheta_n \rangle$ matches the body of the invariant, then last component, by means ϑ_n, must belong to O. Meanwhile $\hat{q}$ means that the sum of all time values presented in the log must belong to this interval, that is $\mathtt{TT}(\langle \vartheta_1, \ldots, \vartheta_n \rangle) \in \hat{q}$.

Definition 7. Let $\vartheta = \langle id_a, \alpha, id_b \rangle$, $\vartheta' = \langle id'_a, \alpha', id'_b \rangle \in \Sigma^{ID}$, be two items of a normalized visible trace, $t \in \mathbb{R}_+$ be a time value and $\hat{p} \in \mathcal{I}_{\mathbb{R}_+}$ be an interval. We define the function compare, denoted by $\mathtt{c} : (\Sigma^{ID} \times \mathbb{R}_+) \times (\Sigma^{ID} \times \mathcal{I}_{\mathbb{R}_+}) \mapsto \mathit{Bool}$ as follow:

$$\mathtt{c}(\langle \vartheta, t \rangle, (\vartheta'/\hat{p})) = ((id_a = id'_a) \wedge (id_b = id'_b) \wedge (\alpha = \alpha') \wedge (t \in \hat{p}))$$

Let $\sigma = \langle \vartheta_1, t_1, \ldots, \vartheta_n, t_n \rangle$, be a normalized visible trace, with $n > 0$, and $\mu = (\vartheta'/\hat{p}, \ldots, \vartheta'_m/\hat{p}_m)$, with $m > 0$, be a body of an invariant. Let $\mathtt{Match} : \mathtt{SetNVT} \times \Phi_{ID}^{body} \mapsto \mathit{Bool}$ be a function that computes if a normalized visible trace and an invariant *matches*. Formally we define $\mathtt{Match}(\sigma, \mu)$ as:

$$\begin{cases} \mathtt{false} & \text{if } n > 1 \wedge m = 1 \vee n = 1 \wedge m > 1 \\ \mathtt{c}(\langle \vartheta_1, t_1 \rangle, (\vartheta'_1/\hat{p}_1)) & \text{if } \sigma = \langle \vartheta_1, t_1, \vartheta_2 \rangle \wedge \mu = (\vartheta'_1/\hat{p}_1) \\ \mathtt{Match}(\langle \vartheta_2, \ldots, \vartheta_n \rangle, (\vartheta'_2/\hat{p}_2, \ldots, \vartheta'_m/\hat{p}_m)) & \text{if } n > 1 \wedge m > 1 \wedge \mathtt{c}(\langle \vartheta_1, t_1 \rangle, (\vartheta'_1/\hat{p}_1)) \\ \mathtt{false} & \text{if } n > 1 \wedge m > 1 \wedge \neg \mathtt{c}(\langle \vartheta_1, t_1 \rangle, (\vartheta'_1/\hat{p}_1)) \\ \mathtt{M'}(\sigma, \mu', \hat{q}, 0) & \text{if } n > 2 \wedge \mu = (\star/\hat{q}, \ldots, \vartheta'_m, \hat{p}_m) \end{cases}$$

where $\mathtt{M'} : \mathtt{SetNVT} \times \Phi_{ID}^{body} \times \mathcal{I}_{\mathbb{R}_+} \times \mathbb{R}_+ \to \mathit{Bool}$ is an auxiliary function used to compute the appearance of the wildcard $\star$.

Let $\mu = (\vartheta'_1/\hat{p}'_1, \ldots \vartheta'_m/\hat{p}'_m)$, with $m > 0$ and $\vartheta'_1 = (id'_a, \alpha', id')$, be the body of an invariant, $\sigma = \langle \vartheta_1, t_1, \ldots, \vartheta_n \rangle$, with $n > 0$ and $\vartheta_1 = (id_a, \alpha, id)$, be a normalized visible trace, $\hat{q} = [q_1, q_2] \in \mathcal{I}_{\mathbb{R}_+}$ be timed interval, and $t \in \mathbb{R}_+$. Formally, we define $\mathtt{M'}(\sigma, \mu, \hat{q}, t)$ as

$$\begin{cases} \mathtt{false} & \text{if } t > q_2 \vee n = 1 \vee (id_a = id'_a \wedge t \notin \hat{q}) \\ \mathtt{Match}(\sigma, \mu) & \text{if } id_a = id'_a \wedge t \in \hat{q} \\ \mathtt{M'}(\langle \vartheta_2, t_2, \ldots, \vartheta_n \rangle, \mu, [q_1, q_2], t + t_1) & \text{if } t \leq q_2 \wedge id_a \neq id'_a \end{cases}$$

Let ϕ be a local invariant and $\sigma = \langle \vartheta_1, t_1, \ldots, \vartheta_n \rangle$ be a trace. We say that σ is *inconclusive* with respect to ϕ if $\forall \sigma' \in \sigma_{<<}$ we have that $\mathtt{Match}(\sigma', \mathtt{Body}(\phi))$ does not hold. Let $O \rhd \hat{q} = \mathtt{Consequent}(\phi)$, we say that σ is *correct* with respect to ϕ if $\forall \sigma' = \langle \vartheta_b, t_b, \ldots, \vartheta_r \rangle \in \sigma_{<<}$, with $1 \leq b < r \leq n$, if $\mathtt{Match}(\sigma', \mathtt{Body}(\phi))$ then we have that $\vartheta_r \in O$ and $\mathtt{TT}(\langle \vartheta_1, t_1, \ldots, \vartheta_r \rangle) \in \hat{q}$.

We say that σ is *not correct* with respect to ϕ if $\exists \sigma' = \langle \vartheta_b, t_b, \ldots, \vartheta_r \rangle \in \sigma_{<<}$, with $1 \leq b < r \leq n$, if $\mathtt{Match}(\sigma', \mathtt{Body}(\phi))$ then we have that $\vartheta_r \notin O$ or $\mathtt{TT}(\langle \vartheta_1, t_1, \ldots, \vartheta_r \rangle) \notin \hat{q}$.

$$\phi_{\mathtt{E1}} = \begin{matrix}\mathtt{E},\mathtt{!Connect},\mathtt{A}[0,39],\\ \mathtt{E},\mathtt{?Hello},\mathtt{G}[0,\infty]\end{matrix} \mapsto \{(\mathtt{E},\mathtt{!Coffee},\mathtt{M})\} \rhd [0,\infty]$$

$$\phi_{\mathtt{E2}} = \mathtt{E},\mathtt{!Connect},\mathtt{A}[0,39], \mapsto \{(\mathtt{E},\mathtt{?Hello},\mathtt{G})\} \rhd [0,39]$$

$$\phi_{\mathtt{E3}} = \begin{matrix}\mathtt{E},\mathtt{!Connect},\mathtt{A}[41,\infty],\\ \mathtt{E},\mathtt{?Hello},\mathtt{G}[0,\infty]\end{matrix} \mapsto \{(\mathtt{E},\mathtt{!Tea},\mathtt{M})\} \rhd [41,\infty]$$

$$\phi_{\mathtt{E4}} = \mathtt{E},\mathtt{!Connect},\mathtt{A}[41,\infty], \mapsto \{(\mathtt{E},\mathtt{?Hello},\mathtt{G})\} \rhd [41,\infty]$$

$$\phi_{\mathtt{E5}} = \begin{matrix}\mathtt{E},\mathtt{!Connect},\mathtt{A}[0,\infty],\\ \mathtt{E},\mathtt{?Hello},\mathtt{G}[0,\infty]\end{matrix} \mapsto \left\{\begin{matrix}(\mathtt{E},\mathtt{!Tea},\mathtt{M})\\ (\mathtt{E},\mathtt{!Coffee},\mathtt{M})\end{matrix}\right\} \rhd [0,\infty]$$

Fig. 3. Local invariants suite for web service E

We denote by $\sigma \diamond \phi$ the fact that σ is correct with respect to ϕ, alternatively $\sigma \neg \diamond \phi$ denotes that is erroneous. □

Next, we will illustrate the semantics of a local invariant by using an example. Let $\phi = id, \alpha, id'/\hat{p}, \star/\hat{p_\star}, id, \alpha', id''/\hat{p}' \mapsto O \rhd \hat{q}$ be a local invariant. This property, with respect to a recorded trace, means that **if** we observe the action α from id to id' in a time belonging to the interval $\hat{p}$, followed by a (possibly empty) sequence of actions without occurrence of the action α', then **if** we observe the input symbol α' from id to id'', and the lapse of time between the performance of the action α and input α' belongs to the interval $\hat{p_\star}$ **then** the head of the invariant must hold. This means that α' must be followed by a triple id-action-id belonging to the set O with an associated time value belonging to $\hat{p}'$. The interval $\hat{q}$ makes reference to the total time that the system must spend to perform the whole trace. Let us remark that an invariant can only detect an error if the body of the invariant, that is the part of the invariant previous to $\mapsto$ symbol, matches the log and, either the functional restriction does not match or any temporal requirement does not match. When an invariant is provided by a tester, before using it to check the correctness of a log, we may ensure that this invariant does not contradict what is represented in the specification model, that is, the invariant has to be *correct* with respect to the specification.

Definition 8. We say that an invariant ϕ is *correct* with respect to a web service $\mathcal{A}$, being $\mathcal{A}_\mathcal{M}$ the LTS associated with $\mathcal{A}$, if the following two conditions hold: For all $\sigma \in \mathtt{NT}(\mathcal{A}_\mathcal{M})$ we have both conditions: or $\sigma \diamond \phi$ or σ is inconclusive with respect to ϕ, and there exits $\sigma \in \mathtt{NT}(\mathcal{A}_\mathcal{M})$ with $\sigma \diamond \phi$. □

Example 4. Let us consider the web services specification presented in Figure 1. Next we show how we can express some properties with timed invariants for the web service `E`. Let us denote that following the same pattern presented for `E`, it is easy to produce the invariants suites for the rest of web services.

The first invariant, by means $\phi_{\texttt{E1}}$, means that on the one hand always that `E` receives `Connect` from `A`, followed by a time value less than or equal to 39 time units, then after sending `Hello` to the web service `G`, it will always receive `Coffee` from `M`; and on the other hand, that the sum of all time values, from `Connect` to `Coffee` is included in $[0, \infty]$.

Another example is the invariant $\phi_{\texttt{E2}}$. It computes on the one hand that always that `E` receives `Connect` from `A`, followed by a time value less than or equal to 39 time units, then it will send `Hello` to the web service `G`, and on the other hand, that the sum of time values from `Connect`, to `Hello` belongs to [0,39].

Let us remark that just in the case of $\phi_{\texttt{E5}}$, there are more than one item in the last set. This mean that the web service is allowed to perform any of these actions after matching its body. □

3.2 Global Invariants

Usually we cannot assume that we have access to the global log. However, we would like to represent some properties involving more than one web service. We call a global log, as a log recorded in a centralized web service, where all actions are marked with the same time-stamp clock, thus there are not measure errors. In this case choreographies help us to check the correctness of these logs, due to the fact that they have a global clock and we can define properties as "local invariants" for the choreography. In our approach we do not consider to have this global log, thus we introduce the notion of *global* invariants, which will help us to represent properties that involve many isolated local logs.

Let us present with our running example, an error produced in local logs that cannot be detected with local invariants. Let us consider the logs presented in the Figure 4. As we can observe, all of them start with the synchronization input `Connect` from the web service `A`. After 40 time units the web services `M` and `E` send `Good Morning` to `C` and `Hello` to `G`(locally 41 time units in `M` and 40.2 time units in `E`). It could be possible that the clocks of the web services `C`, and `G` work slower than the one presented in `M` and `E`; thus, they receive this inputs on 39.8 and on 39.9. After that the web service `M` communicates with the web service `E` and the web service `C` with respect to the web service `G`. As we can observe, taking into account the choreography of our web services, presented in Figure 2. This is not a correct situation of the local logs. The idea is that some web services perform the actions from s_6 to s_8, and the others perform the actions from s_6 to s_{10}. But, as we do not have a global log, only by checking the local behaviours we are unable to see that the set of logs represents an incompatible state.

To solve this problem, first we will define a *global log*, just adding one local log after another local log. The idea is to be able to represent properties over this global log, which help us to detect this kind of errors.

```
M.LOG= ⟨  A.!Connect,41  C.?Good Morning, 36.1,    E.?Tea,    ⟩
E.LOG= ⟨ A.!Connect,40.2      G.?Hello, 36,        M.!Tea,    ⟩
C.LOG= ⟨ A.!Connect,39.8 M.!Good Morning, 50.3, G.?Thank you, ⟩
G.LOG= ⟨ A.!Connect, 39.9     E.!Hello, 50.2,    C.!Thank you, ⟩
```

Fig. 4. Set of local logs recorded in the web services M, E, C and G

Definition 9. Let $\sigma_1 = \langle \vartheta_1^1, \ldots, \vartheta_n^1 \rangle, \ldots, \sigma_j = \langle \vartheta_1^j, \ldots, \vartheta_m^j \rangle$ be j local logs recorded from j different services. We will define a *global log* as the concatenation of these logs, that is $\sigma = \langle \vartheta_1^1, \ldots, \vartheta_n^1, 0, \ldots, 0, \vartheta_1^j, \ldots, \vartheta_m^j \rangle$. We let $\pi_{id}(\sigma)$ denote the projection of σ on a web service identifier id.

Let $\vartheta = (id'_a, \alpha, id)$ be an item of a visible trace, and id_a be a web service identifier. We define the following boolean function: $\mathtt{cmp}(\vartheta, id_a) = (id'_a = id_a)$. Let $\sigma = \langle \vartheta_1, \ldots, \vartheta_n \rangle$ be a global log, and id_a be a web service identifier. Formally, the projection is defined as:

$$\pi_{id_a}(\sigma) = \begin{cases} \langle\rangle & \text{if } n = 1 \wedge \neg \mathtt{cmp}(\vartheta_1, id_a) \\ \langle \vartheta_1 \rangle & \text{if } n = 1 \wedge \mathtt{cmp}(\vartheta_1, id_a) \\ \pi_{id_a}(\langle \vartheta_2, \ldots, \vartheta_n \rangle) & \text{if } \neg \mathtt{cmp}(\vartheta_1, id_a) \\ \langle \vartheta_1, t_1, \rangle \pi_{id_a}(\langle \vartheta_2, \ldots, \vartheta_n \rangle) & \text{if } \mathtt{cmp}(\vartheta_1, id_a) \end{cases}$$

Given two normalized visible traces σ_1, σ_2, we write $\sigma_1 \sim \sigma_2$ if σ_1 and σ_2 cannot be distinguished when making local observations, that is, we have that $\pi_{id}(\sigma_1) = \pi_{id}(\sigma_2)$ for all $id \in ID$. □

In our framework we assume that testers can combine the set of local logs taking into account any criterion. It is easy to proof that any two of them σ_i and σ_j are $\sigma_i \sim \sigma_j$. Following, we will define the notion of global invariants, and the correctness of global logs with respect to them.

Definition 10. Let $\mathcal{D} = (\mathcal{S}, \Sigma, ID, s_0, \{x\}, \mathcal{Z}, \mathcal{T})$ be a choreography. We say that the sequence ψ is a *global invariant* for $\mathcal{D}$, where ψ is defined according to the following EBNF:

$$\psi ::= SET_1 \mapsto_h SET_2$$

In this expression we consider that $h \in \{(1,1), (1,+), (+,+), (+,1)\}$, and SET_1, $SET_2 \subseteq \Phi_{ID}$. Let $\psi = SET_1 \mapsto_h SET_2$ be a global invariant, we will define the functions $\mathtt{SET_1}(\psi) = SET_1$ and $\mathtt{SET_2}(\psi) = SET_2$. Let σ be a global log. We will formally define the semantic of Correct, Incorrect (C/I) or Inconclusive:

h	Verdict	Condition
$\{1,1\}$	C/I	$\exists \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha \rightarrow \exists \phi_\beta \in \mathtt{SET_2}(\psi) : \pi_\beta(\sigma) \diamond \phi_\beta$
$\{1,1\}$	Inconclusive	$\nexists \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha$
$\{1,+\}$	C/I	$\exists \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha \rightarrow \forall \phi_\beta \in \mathtt{SET_2}(\psi) : \pi_\beta(\sigma) \diamond \phi_\beta$
$\{1,+\}$	Inconclusive	$\nexists \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha$
$\{+,1\}$	C/I	$\forall \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha \rightarrow \exists \phi_\beta \in \mathtt{SET_2}(\psi) : \pi_\beta(\sigma) \diamond \phi_\beta$
$\{+,1\}$	Inconclusive	$\exists \phi_\alpha \in \mathtt{SET_1}(\psi)$ such that $\pi_\alpha(\sigma) \neg \diamond \phi_\alpha$
$\{+,+\}$	C/I	$\forall \phi_\alpha \in \mathtt{SET_1}(\psi) : \pi_\alpha(\sigma) \diamond \phi_\alpha \rightarrow \forall \phi_\beta \in \mathtt{SET_2}(\psi) : \pi_\beta(\sigma) \diamond \phi_\beta$
$\{+,+\}$	Inconclusive	$\exists \phi_\alpha \in \mathtt{SET_1}(\psi)$ such that $\pi_\alpha(\sigma) \neg \diamond \phi_\alpha$

$$\psi = \left\{ \begin{array}{c} (\texttt{M},\texttt{!Connect},\texttt{A})/[40,\infty] \\ (\texttt{M},\texttt{?Good Morning},\texttt{C})/[0,\infty] \end{array} \mapsto \{(\texttt{M},\texttt{?Tea},\texttt{E})\} \rhd [40,\infty], \right\} \mapsto_{(+,+)} \left\{ \begin{array}{c} (\texttt{C},\texttt{!Connect},\texttt{A})/[40,\infty] \\ (\texttt{C},\texttt{!Good Morning},\texttt{M})/[0,\infty] \end{array} \mapsto \{(\texttt{C},\texttt{?Good Bye},\texttt{G})\} \rhd [40,\infty], \right\}$$
$$\left\{ \begin{array}{c} (\texttt{E},\texttt{!Connect},\texttt{A})/[40,\infty] \\ (\texttt{E},\texttt{?Hello},\texttt{G})/[0,\infty] \end{array} \mapsto \{(\texttt{E},\texttt{!Tea},\texttt{M})\} \rhd [40,\infty] \right\} \qquad \left\{ \begin{array}{c} (\texttt{G},\texttt{!Connect},\texttt{A})/[40,\infty] \\ (\texttt{G},\texttt{!Hello},\texttt{E})/[0,\infty] \end{array} \mapsto \{(\texttt{G},\texttt{!Good Bye},\texttt{C})\} \rhd [40,\infty] \right\}$$

Fig. 5. Choreography of C, E, G and M

The symbol $\diamond$ used to represent the correctness is overloaded, we will denote by $\sigma \diamond \psi$ that σ is *correct* with respect to ψ. □

Next, we define the correctness of global invariants with respect to both, the orchestration models and the choreography model. Let us remark that we are not allowed with any global clock for checking the temporal restrictions with respect to the choreography. So, on the one hand, we might check the correctness of the time restrictions of the global invariant against the web services and, on the other hand, we might check that the global invariant does not contradict the choreography.

Definition 11. Let $\psi = SET_1 \mapsto_h SET_2$ be a global invariant, and $\mathcal{D} = (\mathcal{S}, \Sigma, ID, s_0, \{x\}, \mathcal{Z}, \mathcal{T})$ be a choreography. We say that ψ is correct if for all $tr \in \texttt{NT}(D)$ we have that:

- Or $tr \diamond \psi$ or tr is inconclusive with respect to ψ, and there exists at least one tr that $tr \diamond \psi$.
- For all $\phi_\alpha \in SET_1 \cup SET_2$ we have that ϕ_α is correct with respect to α.

□

We conclude this section with the proposed problem of our running example. Let us summarize all information that we have, present the problem, and show the solution. We are provided with a set of web services, defined in Figure 1, which are modeled by using an adaptation of timed automata. For this set of web services, we have defined a choreography, also modeled by using a timed automata, and presented in Figure 2. We have introduced a set of local invariants to check the correctness of the web services, see Figure 3. Due to the fact we do not have a global clock, we cannot assume that all internal clocks of the web services work properly. Thus, we could have that some of them work faster than the others. This situation could produce the set of local logs presented in Figure 4. As we discussed, with only local invariants we are not allowed to decide that this set of logs is incorrect. With the use of global invariants, we can detect this fault. Let us consider the global invariant presented in Figure 5. When we check the correctness of the set of logs of the Figure 4 with respect to this invariant, we detect an error on them. Thus, we are able to detect an unexpected behaviour in the composition of these web services.

4 Conclusions and Future Work

In this work we have presented a formal framework to perform passive testing of distributed systems taking into account time information. We assume that we

are provided with a formal specification of both the web services, and the interaction between them. These specifications are modeled by an adaptation of the well known timed automaton model. One contribution of this paper is to define (local)*invariants* for web services. A (local) invariant represents testing-properties, expresses by using input and outputs actions, for checking the correctness of the recorded traces (i.e, logs) of the system.

Another contribution of this paper is to discuss about some errors that are never detected only using local invariants. These errors are based into the idea that each web service has got its own set of local clock, and they do not share these clocks, it means, some of them can be faster that the others. Regarding, in our work we assume that we are not provided with a global clock. This scenario can produce erroneous undetectable situations using only sets of local invariants.

The last contribution of this paper is to provide a way to define (global) *invariants* for a set of web services without having a global clock. To finalize, we discuss that these invariants allow us to detect class of errors that we were unable to detect only using local invariants. As future work, we would like to upgrade our PASsive TEsting tool[1] with this methodology. On the one hand, we will implement the algorithms of checking the correctness of local and global invariants with respect to web services and choreography; and on the other hand algorithms for checking the correctness of logs with respect to local and global invariants.

Acknowledgments

We would like to thank the reviewers of this paper for the careful reading. The quality of the paper has notably increased by considering their useful comments and suggestions.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Andrés, C., Cambronero, M.E., Núñez, M.: Formal passive testing of service-oriented systems. In: 7th Int. Conf. on Services Computing, SCC 2010, pp. 610–613. IEEE Computer Society Press, Los Alamitos (2010)
3. Andrés, C., Merayo, M.G., Núñez, M.: Passive testing of timed systems. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 418–427. Springer, Heidelberg (2008)
4. Andrés, C., Merayo, M.G., Núñez, M.: Supporting the extraction of timed properties for passive testing by using probabilistic user models. In: 9th Int. Conf. on Quality Software, QSIC 2009, pp. 145–154. IEEE Computer Society Press, Los Alamitos (2009)
5. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: Towards automated wsdl-based testing of web services. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 524–529. Springer, Heidelberg (2008)

[1] (https://kimba.mat.ucm.es/paste)

6. Bayse, E., Cavalli, A., Núñez, M., Zaïdi, F.: A passive testing approach based on invariants: Application to the WAP. Computer Networks 48(2), 247–266 (2005)
7. Benharref, A., Dssouli, R., Serhani, M., Glitho, R.: Efficient traces' collection mechanisms for passive testing of web services. Information & Software Technology 51(2), 362–374 (2009)
8. Bentakouk, L., Poizat, P., Zaïdi, F.: A formal framework for service orchestration testing based on symbolic transition systems. In: Núñez, M., Baker, P., Merayo, M.G. (eds.) TESTCOM 2009. LNCS, vol. 5826, pp. 16–32. Springer, Heidelberg (2009)
9. Díaz, G., Rodríguez, I.: Automatically deriving choreography-conforming systems of services. In: 6th IEEE Int. Conf. on Services Computing, SCC 2009, pp. 9–16. IEEE Computer Society Press, Los Alamitos (2009)
10. Frantzen, L., de las Nieves Huerta, M., Kiss, Z.G., Wallet, T.: On-the-fly model-based testing of web services with jambition. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 143–157. Springer, Heidelberg (2009)
11. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal methods to support testing. ACM Computing Surveys 41(2) (2009)
12. Raimondi, F., Skene, J., Emmerich, W.: Efficient online monitoring of web-service SLAs. In: 16th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, FSE 2008, pp. 170–180. ACM Press, New York (2008)
13. Weerawarana, S., Chinnici, R., Gudgin, M., Curbera, F., Meredith, G.: Web services description language (WSDL), Version 2.0, 1 (2004)

On Lifecycle Constraints of Artifact-Centric Workflows*

Esra Kucukoguz and Jianwen Su

Department of Computer Science
University of California at Santa Barbara
{esra,su}@cs.ucsb.edu

Abstract. Data plays a fundamental role in modeling and management of business processes and workflows. Among the recent "data-aware" workflow models, artifact-centric models are particularly interesting. (*Business*) *artifacts* are the key data entities that are used in workflows and can reflect both the business logic and the execution states of a running workflow. The notion of artifacts succinctly captures the fluidity aspect of data during workflow executions. However, much of the technical dimension concerning artifacts in workflows is not well understood. In this paper, we study a key concept of an artifact "lifecycle". In particular, we allow declarative specifications/constraints of artifact lifecycle in the spirit of DecSerFlow, and formulate the notion of lifecycle as the set of all possible paths an artifact can navigate through. We investigate two technical problems: (*Compliance*) does a given workflow (schema) contain only lifecycle allowed by a constraint? And (*automated construction*) from a given lifecycle specification (constraint), is it possible to construct a "compliant" workflow? The study is based on a new formal variant of artifact-centric workflow model called "ArtiNets" and two classes of lifecycle constraints named "regular" and "counting" constraints. We present a range of technical results concerning compliance and automated construction, including: (1) compliance is decidable when workflow is atomic or constraints are regular, (2) for each constraint, we can always construct a workflow that satisfies the constraint, and (3) sufficient conditions where atomic workflows can be constructed.

1 Introduction

Business process management (BPM) has received a rapidly increasing interest in research communities (MIS, CS, and application domains including digital governments, health care delivery, as well as traditional business applications) [5,31]. A key reason is that BPM as a core part of a business enterprise has a wide scope (resource including human management, workflow management, etc.) and is difficult in aspects including discipline barriers among business administration, MIS, IT, etc., and effective management of process/workflow changes. The demand for workflow management tools is enormous. On the other hand, BPM as a research area is rather appealing since there is a lack of a suitable technical framework or model that includes process, data, resources, and human, and can help separating technical problems so that they can be

* Supported in part by NSF grant IIS-0812578 and a grant from IBM.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 71–85, 2011.

addressed individually and independently [5]. In this paper, we introduce integrity constraints for workflow executions and study the interactions of workflow models and such constraints.

Data plays a fundamental role in modeling and management of business workflows [5]. For example, it is quite often that a workflow starts in response to an (external) request that records some vital information about the request; at each step of a workflow execution, proper bookkeeping is made so that the results of actions taken are reflected and can be used for future decisions, and even the actions themselves are logged for system reasons (e.g., reliability) and business reasons (e.g., accountability). Among the recent proposals for "data-aware" workflow models, artifact-centric models [23,9,2] are particularly interesting. Here (*business*) *artifacts* mean the key data entities that are used in workflows and can reflect both the business logic and the execution states of a running workflow. The notion of artifacts succinctly captures the fluidity aspect of data during workflow executions.

Although artifact-centric modeling approach is suitable for applications [3], many challenges remain in developing technical models for artifact centric workflows. Previous studies on artifact-centric workflow models focused on formal models [9,2,1], verification of temporal properties concerning the workflow logic [9,13,6], static analysis of model well-formedness [2], automated construction from non-temporal goals [8], etc. However, there are still many issues concerning artifacts in workflows that are not well understood.

In this paper, we study the key concept of "lifecycle" for artifacts. A lifespan of an artifact is the sequence of services it encountered during its life time. A lifecycle of an artifact class is the set of all possible lifespan by artifacts in the class. In the formal study, we introduce a variant model for artifact-centric workflows called ArtiNet. ArtiNet workflows resemble Petri nets (see [22] for a tutorial) with two key differences. First, artifacts are used in instead of "tokens". Each artifact belongs to a "class" and places are "typed", i.e., each place may store artifacts of a fixed class. Second, when a transition have multiple input (output) places, only one artifact of each input class is consumed (generated) at each firing.

Motivated by DecSerFlow [32,33], we allow declarative specifications of or constraints on artifact lifecycle. We consider two formalisms for lifecycle specifications: *regular* expressions and semilinear sets of Parikh maps [24] that we call *counting* constraints. As lifecycle constraints, we study the *compliance* problem: does a given workflow only contain lifecycle allowed by a constraint? As lifecycle specification, we investigate the *automated construction* problem: from a given lifecycle specification, is it possible to construct a workflow that "realizes" (satisfies) the specification?

DecSerFlow is a declarative language for specifying permitted sequences of services in a workflow. Earlier work on DecSerFlow focused primarily on implementation of DecSerFlow specifications [33], mapping into SCIFF [21], and verification of logical properties [4]. The compliance problem for regular lifecycle constraints was studied earlier [28,18,17,34]. However, the notion of compliance in these works is "syntactic", i.e., based on containment of transition relations. Our model of compliance is semantic and our results naturally generalize the earlier results.

We present a range of technical results concerning compliance and automated construction problems. We show the following. (1) The compliance problem is decidable for either atomic workflows or regular constraints, with the case of workflows and counting constraints remains open. (2) For each regular/counting specification we can always construct a workflow that realizes the constraint, in particular, each regular constraint is realized by an atomic workflow. (3) We also give cases when atomic workflows can be constructed for counting constraints with or without regular constraints.

This paper is organized as follows. Section 2 motivates the problem with an example. Section 3 defines the ArtiNet model, Section 4 introduces the regular and counting constraints. Sections 5 and 6 focus on compliance and automated construction problems, respectively. Section 7 concludes the paper.

2 Motivations

We use a workflow example to motivate artifact lifecycle constraints and the problems studied in this paper. In this example (Fig. 1), preselling refers to selling an apartment before completing the construction. A preselling permit for a group of apartments must be obtained by a developer before selling activities happen. This workflow is simplified from a similar workflow for real estate management in Hangzhou, China [19] where permits are issued by the real-estate administration office of the city.

In Fig. 1, upon receiving an application, an artifact of PAF (Preselling Approval File) is created. For each apartment listed in the PAF, an AF (Apartment File) is created. Each AF is first verified by a compliance officer, and then evaluated by a pricing estimator. Finally an approving process takes the PAF and the reviews for each AF and approves and completes the application.

As a part of the government requirement, each PAF application must have one preliminary review and and one approving service (task), in that order. This can be easily expressed as a regular constraint "$(\textit{receiving App-Form})(\textit{PAF prelim. review})^{+}(\textit{PAF approving})^{+}$". Another constraint to satisfy involves the number of times services are executed. Note that each apartment in a PAF application need to be reviewed separately. Hence the number of *PAF preliminary review* executions should be the same as the number of *PAF approving* executions, otherwise, some apartments could miss the *PAF approving* service before the entire PAF application is approved. Both example properties are properties on PAF artifact "lifecycle".

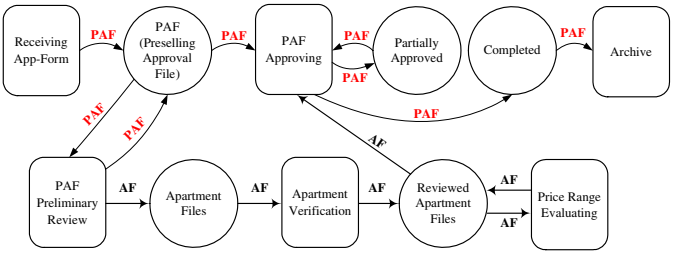


Fig. 1. An ArtiNet Workflow for Apartment Preselling Approval

In this paper we focus on artifact lifecycles, and study the compliance and automated construction problems for workflows in the ArtiNet model. When a new workflow is designed, it is always desirable to know if the workflow is compliant with specified lifecycle constraints. In [28,18,17] authors studied OLCs (object life cycles), and the conformance and coverage problems of OLCs by business process models. A comparison with their results is provided in Section 5. In [34], authors studied the coupling between objects and proposed a method to compute the expected coupling of a process model. Different from these earlier studies, our study also includes automated construction of workflows from lifecycle constraints.

3 ArtiNet: A Formal Model for Artifact-Centric Workflows

In this section, we define the key concepts needed for the technical presentation, including "artifacts", "workflow", "configurations", and "enactments".

Intuitively, artifacts are (abstract) objects that can "flow" through a workflow. The workflow model resembles Petri nets [25] (see e.g., [22]) by substituting tokens with artifacts. Perhaps, the main semantic difference lies in the transition firing rule. But as we will explain below, the firing rule can be simulated by standard Petri nets. Our workflow model is a formal version of ArtiFlow [19] used in the ArtiMT system [20].

In the technical development, we assume the existence of the following countably infinite, pairwise disjoint sets:

- $\mathcal{A}$ of *(artifact) class (names)*,
- $\mathcal{S}$ of services (names),
- $\mathcal{P}$ of *places*, and
- $\mathcal{T}$ of *transitions*.

Definition 1. Let $\boldsymbol{B}$ be a class. A *workflow* for $\boldsymbol{B}$ is a tuple $W = (\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ satisfying all of the following conditions:

- Σ is a finite (possibly empty) set of classes such that $\boldsymbol{B} \notin \Sigma$,
- $P \subseteq \mathcal{P}$ is a finite set of places (or repositories),
- $T \subseteq \mathcal{T}$ is a finite set of transitions and $\tau_s, \tau_f \in T$ are the *seed* and *archival* transitions respectively,
- $E \subseteq P \times (T - \{\tau_s\}) \cup (T - \{\tau_f\}) \times P$ is a set of edges,
- $\boldsymbol{r} : P \to \Sigma \cup \{\boldsymbol{B}\}$ is a mapping that assigns each place $p \in P$ a class such that if $(\tau_s, p) \in E$ or $(p, \tau_f) \in E$ then $\boldsymbol{r}(p) = \boldsymbol{B}$,
- For each transition $\tau \in T - \{\tau_s, \tau_f\}$, $(p, \tau) \in E$ and $\boldsymbol{r}(p) = \boldsymbol{B}$ for some place $p \in P$ iff $(\tau, q) \in E$ and $\boldsymbol{r}(q) = \boldsymbol{B}$ for some place $q \in P$, and
- $\boldsymbol{s} : T - \{\tau_s, \tau_f\} \to \mathcal{S}$ is a mapping that assigns each transition a service name.

The workflow W is *atomic* if $\Sigma = \varnothing$.

If W is a workflow for $\boldsymbol{B}$, we also call $\boldsymbol{B}$ the *focused* class of W. (Σ in W consists of *auxiliary* classes.) To simplify the presentation, for the remainder of this paper, we fix some focused class $\boldsymbol{B}$.

Definition 2. Given a workflow $W = (\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ for a class $\boldsymbol{B}$, a *configuration* of W is a mapping C from P to natural numbers. A configuration is *empty* if it assigns 0 (zero) to every place.

If a place p is assigned $C(p)$, then p stores $C(p)$ number of artifacts.

Let $W = (\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ be a workflow. The set of *input* (resp. *output*) *places* of a transition τ, denoted by $\bullet\tau$ (resp. $\tau\bullet$), is the set of all places that have an edge entering (resp. leaving) the transition τ. We also denote by $(\bullet\tau)_{\boldsymbol{A}}$ (resp. $(\tau\bullet)_{\boldsymbol{A}}$) denote all input (resp. output) places of the transition τ *labeled* $\boldsymbol{A}$. For a set P of places, we define $\boldsymbol{r}(P) = \{\boldsymbol{r}(p) \mid p \in P\}$. In the remaining of this paper, we will frequently use the notations $\boldsymbol{r}(\bullet\tau)$ and $\boldsymbol{r}(\tau\bullet)$ to mean input and output labels of τ, respectively.

Definition 3. Given a workflow $W = (\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ for a class $\boldsymbol{B}$, a transition $\tau \in T$ is *enabled* in a configuration C if for each $\boldsymbol{A} \in \boldsymbol{r}(\bullet\tau)$, $\Sigma_{p \in (\bullet\tau)_{\boldsymbol{A}}} C(p) > 0$.

This condition states that a transition is enabled if there is at least one artifact available from each input class from all input places combined. A transition can have multiple input places with the same label, but only one artifact from each input class is sufficient to enable the transition. Note that this is different from standard Petri nets that consume a token from each input place. However, one can simulate this semantics in Petri nets. For example, if a transition τ has input places p, q storing artifacts of $\boldsymbol{B}$. We could construct an additional place p' and two transitions τ_p and τ_q that reads input from p and q (resp.) and outputs to the place p'. We also modify τ's input so that it takes one input from p'. Thus, the "picking an artifact from *either* places" construct in our model is turned into "picking *some* artifact from a place" in Petri nets.

A workflow "moves" from one configuration to another when a transition fires. Given a configuration, there might be more than one possible transitions that are enabled. One of the enabled transitions is chosen to be fired nondeterministically.

When a transition fires, it consumes one artifact from each input class. It generates one artifact for each output class and puts it nondeterministically in one of the output places if there are multiple possible places to put the artifact.

The firing of a transition τ is defined as a triple (C, τ, C'). It indicates transition τ is invoked and the workflow moves from configuration C to configuration C'.

Definition 4. Given a workflow $W = (\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$, two configurations of C_1, C_2 of W and a transition $\tau \in T$, C_1 *derives* C_2 *using* τ if the following conditions are all true:

1. τ is *enabled* in C_1,
2. If $\boldsymbol{A}$ is an input class and let $p_{\boldsymbol{A}}$ be the place artifact labeled $\boldsymbol{A}$ is chosen to be consumed, then for each input place $p \in (\bullet\tau)_{\boldsymbol{A}}$,

$$C_2(p) = \begin{cases} C_1(p) - 1 & \text{If } p = p_{\boldsymbol{A}} \\ C_1(p) & \text{otherwise} \end{cases}$$

3. If $\boldsymbol{A}$ is an output class and let $p_{\boldsymbol{A}}$ be the place artifact labeled $\boldsymbol{A}$ is put then for each output place $p \in (\tau\bullet)_{\boldsymbol{A}}$,

$$C_2(p) = \begin{cases} C_1(p) + 1 & \text{If } p = p_{\boldsymbol{A}} \\ C_1(p) & \text{otherwise} \end{cases}$$

Definition 5. An *enactment* of a workflow $(\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ is an alternating sequence of configurations and transitions $C_0\tau_0C_1\tau_1C_2\cdots\tau_{n-1}C_n$ such that

1. C_0 is the empty configuration,
2. $\tau_0 = \tau_s$ (seed transition) and for all $1 \leqslant i < n$, $\tau_i \in T - \{\tau_s\}$, and
3. For each $0 \leqslant i < n$, C_i derives C_{i+1} using τ_i.

The enactment is *complete* (or *terminating*) if the last configuration C_n is empty and the last transition is the archival transition.

4 Lifecycle Constraints of Artifacts

In this section, we introduce a notion of integrity constraints for ArtiNet workflows, called "lifecycle constraints". Intuitively, lifecycle constraints limits the way how workflow executions should be carried out. In artifact-centric workflows, a workflow enactment (or instance) executes a collection of services in certain order to accomplish the business goal. The constraints in our model thus focus on the sequencing aspect. In the general case, lifecycle constraints could involve the contents of artifacts. However in this initial study, we focus on constraints that do not examine data values. In this section we formulate the key concepts, the technical discussions will be presented in the next two sections.

Clearly, the language DecSerFlow [32,33] for specifying sequencing of service orderings is easily seen as a class of lifecycle constraints. In [29,30], an algebra was developed for event sequences. Algebraic expressions were used as constraints on task sequences during the execution. In [18,28,17] authors studied the consistency of OLC (object life cycle) with respect to a business process model. Finite state machines are used to represent OLCs. In that aspect, our definition of lifecycle coincides with the OLC notion in these papers.

Definition 6. Let $W{=}(\Sigma, P, T, \tau_s, \tau_f, E, \boldsymbol{r}, \boldsymbol{s})$ be a workflow for artifact class $\boldsymbol{B}$ and $C_0\tau_0C_1\tau_1C_2\cdots\tau_{n-1}C_n$ a complete enactment of W. A *lifespan of* $\boldsymbol{B}$ *in* W is a sequence of service names corresponding to the transitions in the complete enactment: $\boldsymbol{s}(\tau_0)\boldsymbol{s}(\tau_1)\cdots\boldsymbol{s}(\tau_{n-1})$. The *lifecycle* $L(W)$ *of* $\boldsymbol{B}$ is the set of all lifespans of $\boldsymbol{B}$ in W.

Note that the seed and archival transitions are ignored in a lifespan. Let W be a workflow. We define $\boldsymbol{S}_W$ as the set of all services that are images of the service mapping in W.

Definition 7. Let W be a workflow for a class $\boldsymbol{B}$. A (*life-cycle*) *constraint on* $\boldsymbol{B}$ is a subset of $\boldsymbol{S}_W^*$, i.e., a (possibly infinite) set of words over $\boldsymbol{S}_W$. The workflow W *satisfies* (or *realizes*) a life-cycle constraint γ, if $L(W) \subseteq \gamma$ (resp. $L(W) = \gamma$).

Let W be a workflow for class $\boldsymbol{B}$. We consider two classes of life-cycle constraints: "regular" and "semi-linear" constraints. A constraint γ on $\boldsymbol{B}$ is *regular* if γ is a regular language over $\boldsymbol{S}_W$. In this paper, we further assume that regular constraints are specified in form of regular expressions [14]. Assuming $a, b, c \ldots$ are services in $\boldsymbol{S}_W$ examples of regular constraints include: $(a + b)^*c$, $(ab^*a)^*$, and $a^*b^*(c + d)$.

To define the second class of constraints, we first assume some fixed enumeration $s_1, ..., s_n$ of $\boldsymbol{S}_W$. The *Parikh map* [24] of a word $w \in \boldsymbol{S}_W^*$ is a vector of natural numbers $\mathit{Parikh}(w) = (v_1, ..., v_n)$ such that for each $i \in [1..n]$, v_i is the number of occurrences of s_i in w. Let $\gamma \subseteq \boldsymbol{S}_W^*$ be a set of words over $\boldsymbol{S}_W$. The *Parikh map* of γ is the set $\mathit{Parikh}(\gamma) = \{\mathit{Parikh}(w) \mid w \in \gamma\}$.

We say that $\gamma \subseteq \boldsymbol{S}_W^*$ is a *counting* constraint if its Parikh map $\mathit{Parikh}(\gamma)$ is a semilinear set [10]. Recall that a linear set is a pair $(\bar{v}, P)$ where $\bar{v}$ is an n-vector and $P = \{\bar{v}_1, ..., \bar{v}_n\}$ is a finite set of *base* vectors. It defines the set of points in n-dimensional space $\{\bar{u} \mid \bar{u} = \bar{v} + \sum_{i=1}^{n} k_i \bar{v}_i$, where $k_i \geqslant 0$ for all $1 \leqslant i \leqslant n\}$. A semilinear set is a finite union of linear sets.

An example of a counting constraint is $i \cdot (1, 2, 0) + j \cdot (0, 0, 2)$ where $i, j \geqslant 0$. This constraint states that s_2 occurs twice as much as s_1 and s_3 occurs an even number of times.

Lemma 4.1. Let W be a workflow for a class $\boldsymbol{B}$ and Γ be a finite set of regular (resp. counting) constraints on $\boldsymbol{B}$. Then there exists a regular (resp. counting) constraint γ such that the following are equivalent:

1. W satisfies every constraint in Γ.
2. W satisfies the constraint γ.

The above lemma easily follows from the fact that both regular languages and semilinear sets are closed under intersection [14,10].

We conclude the section with the following results.

Proposition 4.2. (1) The lifecycle of every atomic workflow is a regular language.
(2) The lifecycle of a workflow is always context-sensitive, and there are workflows whose lifecycles are (i) context-free but not regular, or (ii) not context-free.

Item 1 is rather straightforward. Item 2 is not surprising, since (formal) languages defined by Petri nets are always context-sensitive [26,22].

5 Compliance of Lifecycle Constraints

In this section, we focus on the "compliance" problem for ArtiNet workflows and lifecycle constraints. The problem easily occurs in practice as one would ensure if the existing workflow would satisfy the constraints. We show that the problem is decidable for atomic workflows with either type of constraints, and for workflows with regular constraints. The case of workflows with counting constraints is left open. We also study compliance of DecSerFlow constraints.

Lifecycle Compliance (LC) Problem: Given a workflow W and a regular or counting constraint γ, determine if W satisfies γ.

Unlike the work in [29,30] where enforcement is done at runtime, we study the static analysis problem of deciding if the workflow will always satisfy the constraints. The object lifecycle compliance problem studied in [28,18,17] is closely related to the compliance problem defined above. However, the notion of compliance in [28,18] requires that the object state transitions in the business process is a subset of the transitions in

a specified lifecycle. This is a stronger requirement, i.e., if a workflow is compliant in our model, it may not be compliant by their definition. However, compliance holds in their model would imply compliance in our model. [17] studied compliance considering side-effects while our model does not consider side-effects.

The main results of the section are now stated below.

Theorem 5.1. LC problem is decidable for atomic workflows. For the general case of workflows, LC is decidable for regular constraints.

ArtiNet workflows can be reduced to Petri nets, which as language acceptors, accept context-sensitive languages [22,26]. Since the emptiness problem for context-sensitive languages is undecidable [14], one cannot directly apply known results for the LC problem. On the other hand, Petri nets can be converted to "partially blind multicounter machines" [16], and it is shown [16] that the containment of a Petri net language in a regular language can be determined.

In the remainder of this section, we focus on atomic workflows.

Lemma 5.2. Given an atomic workflow W and a regular constraint γ, it is decidable to check if W satisfies γ.

To establish the lemma, it is sufficient to note that one can effectively convert each regular expression γ to an *equivalent* (non-deterministic) finite state machine M_γ. By Proposition 4.2, the language accepted by W is regular; subsequently, one can construct a finite state machine M_W that accepts the language $L(W)$. It follows that W satisfies γ iff $M_W \subseteq M_\gamma$ (with an abuse of notation). The latter is known to be decidable.

The LC problem for atomic workflows and regular constraints is however PSPACE-complete. This follows from the fact that checking if a non-deterministic FSA accepts Σ^* is PSPACE-hard [11]. We note here that the main source of complexity is from constraints and not workflows. To see its membership in PSPACE, we note that the containment $M_1 \subseteq M_2$ of two FSAs can be reduced to checking the emptiness of $M_1 \cap M_2^c$ (M_2^c is the complement of M_2). Although M_2^c has an exponential size, the emptiness checking can be done without writing down the complete complement machine.

Lemma 5.3. Let γ be a regular constraint that is equivalent to a deterministic FSA M_γ. It can be determined in $O(n^2)$ time if an atomic workflow satisfies γ, where n is the size of the workflow and M_γ.

Clearly, the workflow can be viewed as an FSA M_W. $M_W \subseteq M_\gamma$ iff $M_W \cap M_\gamma^c$ is empty. Since M_γ is deterministic, M_γ^c and M_γ have the same size. Finally, the emptiness of intersection of two FSAs can be determined in the size of their product.

An interesting application of Lemma 5.3 is on compliance of DecSerFlow [32,33] constraints by ArtiNet workflows. DecSerFlow is a declarative language for specifying permitted sequences of tasks in a workflow. DecSerFlow is based on LTL [7] with restrictions in syntax and semantics.

A DecSerFlow specification includes a finite set of services (tasks) and a set of constraints. It defines a set of sequences of services permitted in workflow execution. There are two types of constraints. The first type is cardinality constraints that are associated with individual services, which limits the number of invocations allowed for a service (i.e., occurrences of the service in the sequence). The second type of constraints are

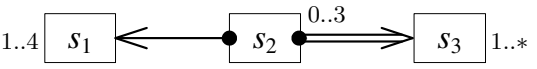
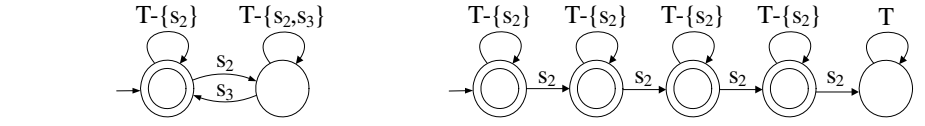


Fig. 2. A DecSerFlow Specification



(a) Alternate response DecSerFlow constraint (b) DecSerFlow cardinality constraint

Fig. 3. FSAs corresponding to DecSerFlow constraints

relational constraints. A relational constraint can be associated with two services and limits occurrences and/or ordering of the involves services.

Fig. 2 shows a simple DecSerFlow specification with three services and two relational constraints. The cardinality constraints restrict the lower and upper bounds of the numbers of occurrences of services ($*$ means unbounded). The relational constraint from s_2 to s_1 requires that each s_2 must be followed by some s_1, i.e., s_2 cannot occur as the last task. The constraint from s_2 to s_3 requires that between two occurrences s_2 there must be an s_3 and that some s_3 must appear after the last s_2.

There are 11 types of relational constraints in DecSerFlow and some of them have negated versions [32, 33]. For example, *existence-response* on two services s_1, s_2 means that if s_1 executes, s_2 should also execute but the timing of these executions can be arbitrary; the relational constraint *response* from s_1 to s_2 requires that whenever s_1 executes, there is a "responding" execution of s_2 *later on*. In this case, timing is important. However, s_2 does not have to execute immediately after s_1, and two or more executions of s_1 may share the same s_2 responding execution. The dual relational constraint *precedence* from s_1 to s_2 enforces that if s_2 executes there must be a preceding s_1 execution.

We can easily view each DecSerFlow specification as a lifecycle constraint. In this sense, the following can be shown.

Theorem 5.4. Let γ be a DecSerFlow constraint and W a workflow. It can be decided in $O(nml)$ time if W satisfies γ, where n is the size of W, m the total number of services and relational constraints in γ, and l the largest integer in the cardinality bounds in γ.

In Fig. 3(a), we give the FSA corresponding the *alternate response* constraint in DecSerFlow between s_2 and s_3 showed in Fig. 2. (T stands for the finite set of services.) In this FSA, after each s_2 occurrence, there has to be at least one s_3 before the next s_2 occurrence. Therefore, this FSA is exactly the *alternate response* constraint. In Fig. 3(b), we give the FSA corresponding to the cardinality constraint on s_2 in Fig. 2, where s_2 can occur at most 3 times.

For each relational constraint γ' in γ, we can easily construct a deterministic FSA with a constant number of states. By Lemma 5.3, satisfaction of γ' can be tested in $O(n)$ time. For each cardinality constraint, we can also construct a deterministic DFA with at most l number of states. The theorem follows easily. We now turn to counting constraints.

Lemma 5.5. Given an atomic workflow W and a counting constraint γ, it can be determined if W satisfies γ.

From Proposition 4.2, the lifecycle of W is a regular language. To establish decidability in Lemma 5.5, we note that the Parikh map of each regular language is semilinear [15]. It is also known that semilinear sets are closed under intersection, union, and complement [10]. Thus, W satisfies γ iff $\mathit{Parikh}(L(W)) \cap \overline{\gamma}$ is empty ($\overline{\gamma}$ denotes the complement of γ), the latter is decidable [12]. (An alternative is to construct a Presburger formula that states the containment $\mathit{Parikh}(L(W)) \subseteq \gamma$; the decidability follows from the decidability of Presburger arithmetic [27].)

6 Workflow Construction from Lifecycle Specification

In this section, we treat a lifecycle constraint as a workflow specification and study the problem of constructing a workflow from a given constraint. It is obvious that one can always construct an atomic workflow for each regular constraint. Therefore, we restrict our attention to (atomic) workflows and counting constraint or in conjunction with a regular constraint, and study the problem of whether from a given counting constraint we can construct an (atomic) workflow. We show that for each counting constraint we can effectively construct a workflow that realizes (satisfies) the constraint. We also exhibit various cases when atomic workflows can be constructed for a counting constraint.

Automated Construction Problem: Given a counting constraint γ, can we find an (atomic) workflow that realizes (satisfies) γ?

Theorem 6.1. (a) For every regular constraint γ, there is an atomic workflow that realizes γ.
(b) For every counting constraint γ, there is a workflow that realizes γ.

Part (a) of Theorem 6.1 is straightforward. In particular, γ can be converted into a nondeterministic FSA M_γ, and then one can construct an atomic workflow by creating a transition node for each transition in M_γ and a place node for each state in M_γ.

Since each DecSerFlow constraint can be expressed as a regular constraint, the following holds.

Corollary 6.2. Each DecSerFlow constraint can be realized by an atomic workflow.

For Part (b), we use auxiliary artifacts to help counting. For one linear set counting constraint, we can construct one workflow that realizes (satisfies) the constraint. If the counting constraint γ is semilinear, i.e., a finite union of linear sets, we can construct a workflow for each linear set and then combine the workflows into a single workflow.

To demonstrate the key idea and techniques, consider a counting constraint $2z_a = z_b$ that states the number of executions of transition b should be twice as much as the number of occurrences of transition a. The Parikh map of the constraint is $\{(i, 2i) \mid i \geqslant 0\}$, i.e., the base vector is $(1, 2)$. To realize this constraint, in addition to our focused artifact ***B*** we also have one auxiliary artifact ***A***. There is one place for class ***B***, all transitions labeled a or b take a ***B*** artifact from and put it back to the place when they are executed. We have 4 places for the auxiliary artifact ***A*** each representing the vectors

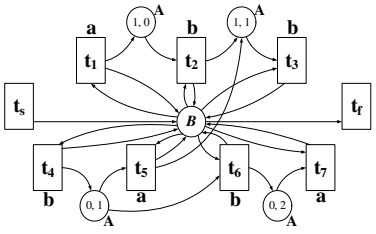


Fig. 4. Realization of a Counting Constraint

(i, j), where $0 \leqslant i \leqslant 1$ and $0 \leqslant j \leqslant 2$, except for $(0, 0)$ (not started yet) and $(1, 2)$ (completed). Each of the a transitions takes an $\boldsymbol{A}$ artifact from (i, j) for some i, j and after execution puts it into $(i+1, j)$ (or just archive it), therefore simulating the increase in the number of a's. Similarly, each of the b transitions should increase the b counts by 1 while leaving a count unchanged. It is clear that the constraint on the numbers of a's and b's has to be satisfied in complete enactments. Fig.4 shows the workflow constructed for the counting constraint $2z_a = z_b$. In a complete enactment of this workflow, all the places should be empty at the end. Therefore, all tokens of the auxiliary artifact $\boldsymbol{A}$ should already be consumed. It can be clearly seen in the figure that when all tokens of $\boldsymbol{A}$ is consumed, the number of executions of transitions b is exactly the twice as much as the number of executions of transition a.

We now describe the construction of a workflow from a counting constraint. We start from linear sets with a single base vector $(m_1, ..., m_k)$, where k is the number of services. We create place nodes for class $\boldsymbol{A}$ for each vector $(n_1, ..., n_k)$ where $0 \leqslant n_i \leqslant m_i$ for each $0 \leqslant i \leqslant k$, except the two vectors $(0, ..., 0)$ and $(m_1, ..., m_k)$. For each place p representing the vector $(n_1, ..., n_k)$ where $\exists j, n_j = 1, n_r = 0$ if $r \neq j, 0 \leqslant j, r \leqslant k$, we create an *initialization* transition τ and an edge (τ, p). For each place p representing the vector $(n_1, ..., n_j, ..., n_k)$, if there is a place p' with the vector $(n_1, ..., n_{j-1}, n_j + 1, n_{j+1}, ..., n_k)$, then we create a *continuation* transition τ and two edges (p, τ), and (τ, p'). For each place p representing the vector $(n_1, ..., n_k)$ where $\exists j, n_j = m_j - 1, n_r = m_r$ if $r \neq j, 0 \leqslant j, r \leqslant k$, we create an *ending* transition τ and an edge (p, τ). In fig.4, created *initialization* transitions are t_1 and t_4, *continuation* transitions are t_2, t_5, t_6, and *ending* transitions are t_3 and t_7. In addition, we create a place node p_* for the focused artifact $\boldsymbol{B}$ and edges (τ_s, p_*) and (p_*, τ_f). We also create edges (τ, p_*) and (p_*, τ) where $\tau \neq \tau_s, \tau_f$. It can be shown that the Parikh map constraint $\{i \cdot (m_1, ..., m_k) \mid i \geqslant 0\}$ is satisfied. When a linear set has more than one base vector, the construction would repeat for each base vector, except that the common place p_* for the focused class is shared. If the counting constraint γ is a finite union of linear sets, we will construct a workflow with disjoint places for linear set and let the seed transition to (randomly) pick one $\boldsymbol{B}$ place to start.

In the remainder of the section, we study the problem of constructing atomic workflows: Given a counting constraint γ, find an atomic workflow that satisfies γ.

Clearly, if the γ is a regular language, it reduced to Theorem 6.1(a). We assume that γ is not regular. Actually, the class of counting constraints includes all context-free languages [24]. Naturally, we are looking for atomic workflows whose lifecycles are sublanguages of γ.

Since atomic workflows are basically FSAs, in the following discussions, we focus on finding regular sublanguages called "factors" rather than constructing workflows.

Definition 8. Given two languages L_1 and L_2, L_2 is said to be a *factor* of L_1 if $L_2 \subseteq L_1$, L_2 is infinite and regular.

Notion of factor naturally leads to the following strategy for automated constructions. Given a constraint γ. Let $L_1, ..., L_k$ be factors of γ, we construct a FSA (workflow) for L where $L = \cup_{1 \leqslant i \leqslant k} L_i$.

Lemma 6.3. The language $L \subseteq \{a, b\}^*$ such that *Parikh*$(L) = \{(i, i) \mid i \geqslant 0\}$ has infinitely many pairwise incomparable factors.

To prove the above lemma, we construct the sets E_n's ($n \geqslant 1$) as follows. $E_n = \{w \mid$ *Parikh*$(w) = (n, n)$ and $w \notin \cup_{i=1}^{n-1} E_i^{\oplus}\}$, where $E_i^{\oplus} = \{w^j \mid w \in E_i$ and $j \geqslant 1\}$.

Claim 1: For each $n \geqslant 1$, E_n is not empty.

Claim 1 holds because there are no words in $\cup_{i=1}^{n-1} E_i^{\oplus}$ that starts with n a's. However, $a^n b^n \in E_n$.

Claim 2: For each $i \geqslant 1$, and each $w \in E_i$, $w^* \subseteq L$.

One can easily show that *Parikh*$(w^*) \subseteq \{(i, i) \mid i \geqslant 0\}$; therefore $w^* \subseteq L$.

Claim 3: For each pair of words $w_1, w_2 \in \cup_i E_i$, if $w_1 \neq w_2$ then $w_1^* \nsubseteq w_2^*$.

If we can find a word $w \in w_1^*$ and $w \notin w_2^*$, then $w_1^* \nsubseteq w_2^*$. Consider $w = w_1$.

If $|w_1| < |w_2|$, then there are no words in w_2^* with the same length as w_1, since each word in the set w_2^* has length $|w_2|$ or greater. Therefore, $w_1 \notin w_2^*$.

If $|w_1| > |w_2|$, to satisfy the condition $w_1 \in w_2^*$, we should have $w_1 = w_2^i$ for some $i \geqslant 1$. This violates our construction, hence such w_1 cannot exists.

If $|w_1| = |w_2|$, only word in w_2^* that has the same length with w_1 is w_2. But obviously, $w_1 \neq w_2$, therefore $w_1 \notin w_2^*$.

Lemma 6.3 naturally generalizes to the following.

Theorem 6.4. Let Σ be a (finite) alphabet. Every non-regular counting constraint over Σ^* has infinitely many pairwise incomparable factors.

L has at least one non-regular subset whose Parikh map is a linear set. Let's assume L' is such a subset of L. Since L' is not regular, there is at least one base vector for the Parikh map of L' that defines a non-regular language. The corresponding non-regular constraint can be written as $\{i \cdot (m_a, m_b, \ldots, m_z)\}$ and at least two of the m_is are non-zero. If at most one of them is non-zero, then it defines a regular language. For the basis vector $\{i \cdot (m_a, m_b, \ldots, m_z)\}$, one can come up with infinitely many pairwise incomparable factors, using a similar construction as shown in Lemma 6.3. In fact, these factors can be constructed as: for each $i \geqslant 1$, $(a^{i \cdot m_a} b^{i \cdot m_b} \cdots z^{i \cdot m_z})^*$. Since L' is a subset of L and L' has infinitely many pairwise incomparable factors, so does L.

Theorem 6.4 shows even if the constraints define non-regular languages, "compliant" implementations by atomic workflows are still possible. And in fact, there are many ways to choose from. However, the following result shows that combining regular and counting constraints may sometimes prohibit atomic workflows.

Lemma 6.5. $L = \{a^n b^n \mid n \geqslant 0\}$ has no factors.

Assume L has a factor L'. Let M be the FSA that accepts the language L'. Since L' is infinite, there exists a word $w \in L'$ such that $w = a^i b^i$ and $|w| >$ number of states in M. By the pumping lemma for regular languages, M must include words not in L (thus not in L').

In spite of the above negative result, in the following, we show that it may still be possible to find atomic workflows for a pair of regular and counting constraints.

Lemma 6.6. Let γ_r and γ_c two a regular and counting (resp.) constraint. if γ_r has no union and one star, then $\gamma_r \cap \gamma_c$ defines a regular language.

Intuitively, the above lemma holds since for each linear set, and each union-free regular expression with at most one star, their intersection must be regular. It is because with no stars, the language is finite and so is the intersection. With one star, the Parikh map of the corresponding regular language can be expressed linear equations with the number i of iterations as a parameter. The intersection is thus a linear set and is either finite or infinite including all i's greater than some fixed number.

With this positive result, it is interesting to obtain more sufficient conditions for factor existence in presence of a regular and a counting constraint.

7 Conclusions

In this paper we provide a formal analysis on artifact-centric workflows, centering around the notion of artifact lifecycle. Although our work is inspired by DecSerFlow, the technical problems examined provide new insights into the interplay between constraints/specifications and workflow models. On one hand, compliance problems are solvable in case of workflow and regular expressions (but remain open for counting constraints). On the other hand, construction of workflows can always be done. Our results do not provide a complete characterization of complexity for the technical problems. There are many open problems, including: compliance of counting constraints by workflows, sufficient (and/or necessary) conditions for existence of atomic workflows for counting constraints with or without regular constraints.

One interesting remark is that while our workflow model is closely related to Petri nets, our study of the technical problems uses a myriad of tools including formal languages, automata, linear algebra, and logic.

References

1. Abiteboul, S., Segoufin, L., Vianu, V.: Modeling and verifying active xml artifacts. Data Engineering Bulletin 32(3), 10–15 (2009)
2. Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)
3. Chao, T., Cohn, D., Flatgard, A., Hahn, S., Linehan, M.H., Nandi, P., Nigam, A., Pinel, F., Vergo, J., Wu, F.Y.: Artifact-based transformation of ibm global financing. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 261–277. Springer, Heidelberg (2009)

4. Chesani, F., Mello, P., Montali, M., Torroni, P.: Verification of choreographies during execution using the reactive event calculus. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 55–72. Springer, Heidelberg (2009)
5. NSF Workshop: Research Challenges on Data-Centric Workflows (May 2009), http://dcw2009.cs.ucsb.edu
6. Deutsch, A., Hull, R., Patrizi, F., Vianu, V.: Automatic verification of data-centric business processes. In: Proc. Int. Conf. on Database Theory (ICDT), pp. 252–267 (2009)
7. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B ch. 7, pp. 995–1072. North Holland, Amsterdam (1990)
8. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In: Proc. Int. Conf. on Database Theory, ICDT (2009)
9. Gerede, C.E., Bhattacharya, K., Su, J.: Static analysis of business artifact-centric operational models. In: IEEE International Conference on Service-Oriented Computing and Applications (2007)
10. Garey, M., Johnson, D.: Computers and Intractability A Guide to the theory of NP-Completeness. Freeman, New York (1979)
11. Ginsburg, S.: The Mathematical Theory of Context Free Languages. McGraw-Hill, New York (1966)
12. Ginsburg, S., Spanier, E.: Bounded Algol-like languages. Transactions of AMS 113, 333–368 (1964)
13. Gerede, C.E., Su, J.: Specification and verification of artifact behaviors in business process models. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 181–192. Springer, Heidelberg (2007)
14. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
15. Ibarra, O.: Private communications (2010)
16. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25, 116–133 (1978)
17. Küster, J.M., Ryndina, K.: Improving inconsistency resolution with side-effect evaluation and costs. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 136–150. Springer, Heidelberg (2007)
18. Küster, J., Ryndina, K., Gall, H.: Generation of BPM for object life cycle compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 165–181. Springer, Heidelberg (2007)
19. Liu, G., Liu, X., Qin, H., Su, J., Yan, Z., Zhang, L.: Automated realization of business workflow specification. In: Proc. Int. Workshop on SOA, Globalization, People, and Work (2009)
20. Lü, Y., Qin, H., Li, J., Chen, Y., Zhang, L., Su, J.: Design and implementation of artimt: An artifact-centric Workflow Management System (2010) (manuscript)
21. Montali, M., Chesani, F., Mello, P., Storari, S.: Towards a decserflow declarative semantics based on computational logic. Technical report, LIA Technical Report 79 (2007)
22. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4) (1989)
23. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)
24. Parikh, R.: On context-free languages. Journal of the ACM 13, 570–581 (1966)
25. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall Inc., Englewood Cliffs (1981)
26. Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: Proc. of IFIP Congress 62, pp. 386–390. North Holland Publ. Comp., Amsterdam (1963)

27. Presburger, M.: über die Vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In: Comptes rendus du premier Congrès des Mathématiciens des Pays Slaves, pp. 92–101, Warszawa (1929)
28. Ryndina, K., Küster, J., Gall, H.: Consistency of business process models and object life cycles. In: Proc. 1st Workshop Quality in Modeling (2006)
29. Singh, M.: Semantical considerations on workflows: An algebra for intertask dependencies. In: Proc. Workshop on Database Programming Languages, DBPL (1995)
30. Singh, M.P., Meredith, G., Tomlinson, C., Attie, P.C.: An event algebra for specifying and scheduling workflows. In: Proc. 4th Int. Conf. on Database Systems for Advanced Applications, DASFAA (1995)
31. van der Aalst, W.M.P.: Business process management demystified: A tutorial on models, systems and standards for workflow management. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 21–58. Springer, Heidelberg (2004)
32. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: Proceedings of Workshop on Web Services and Formal Methods (2006)
33. van der Aalst, W.M.P., Pesic, M.: Specifying and monitoring service flows: Making web services process-aware. In: Test and Analysis of Web Services, pp. 11–55. Springer, Heidelberg (2007)
34. Wahler, K., Küster, J.M.: Predicting coupling of object-centric business process implementations. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 148–163. Springer, Heidelberg (2008)

Conformance Verification of Privacy Policies

Xiang Fu

Department of Computer Science
Hofstra University
Xiang.Fu@hofstra.edu

Abstract. Web applications are both the consumers and providers of information. To increase customer confidence, many websites choose to publish their privacy protection policies. However, policy conformance is often neglected. We propose a logic based framework for formally specifying and reasoning about the implementation of privacy protection by a web application. A first order extension of computation tree logic is used to specify a policy. A verification paradigm, built upon a static control/data flow analysis, is presented to verify if a policy is satisfied.

1 Introduction

The importance of protecting personal information privacy has been recognized for decades (e.g., see the the Health Insurance Portability and Accountability Act [23]). To increase customer confidence, many websites publish their privacy policies regarding the use and retention of information, using standards such as P3P [30] and EPAL [26]. For example, an online store web application can state that the credit card information she collects is used for the purpose of financial charge only and will be destroyed once the transaction is completed.

A publicly stated policy, however, may be violated by malperformed business practices and web application implementations, e.g., saving the credit card information to a "secrete database" instead of destroying it. We are interested in the following problem:

Given a privacy policy which specifies the purpose, retention, and recipient of information, can designers employ static analysis techniques to verify if a web application (including its code level implementation and system configuration) satisfies the policy?

The problem is essentially a verification problem (which we dubbed as "conformance verification"), because this is about checking if a system implementation respects a public specification (e.g., to observe a communication protocol [24,5]).

We propose PV (Privacy Verification), a framework built upon the first order relational logic [18]. The conformance check is semi-automatic and it consists of the following stages: (1) **Modeling**: A web application is modeled as a reactive system that responds to requests from its customers and stakeholders. Each servlet is modeled as an atomic transition rule that updates the knowledge of entities about private information. Here, an entity can be used to describe any "live being" in the model, e.g., a servlet, an employee, a database, an operating

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 86–100, 2011.

system call, etc. (2) **Static Analysis**: To ensure the precision of a model, a static analysis is used for extracting the information flow between entities of a web application, which is later used to construct the formal model of each web servlet. The analysis is conservative in that it may have false positives, but every possible information flow path in a servlet is reported. (3) **Verification**: a privacy policy is specified using a temporal logic, by adapting CTL [10] with first order relational logic components. The verification has to be limited to a finite model for ensuring decidability. The Alloy Analyzer [17] is used in the proof of the concept experiments. In the future, the Kodkod constraint solver [28] can be directly used for discharging symbolic constraints.

The contributions of this paper include the following: (1) PV provides a compact representation of knowledge ownership and its dynamic changes with system execution; (2) Instead of deriving PV from a top-down design, we propose a static analysis framework that extracts PV model from code level implementation. This would save tremendous time in modeling and can better accommodate the evolution of software systems; and (3) the verification paradigm includes a processing algorithm that handles highly expressive CTL-FO, which is not originally available in Alloy (as a model finder).

The rest of the paper is organized as follows. §2 briefly overviews P3P, which motivates the formal model in §3. §4 presents the verification algorithm. §5 introduces the static analysis algorithm that extracts information flow. §6 discusses related work and §7 concludes.

2 Overview of Security Policies

There are several competing standards for privacy protection, e.g., EPAL [26] and P3P [30]. Although often under debate and criticism, P3P has gained wide acceptance. Each P3P security policy consists of a collection of security statements. A statement declares the purpose of the data collection activity, the data group to be collected, the intended retention, the recipient, and the consequence.

Figure 1 shows one sample P3P statement for a web application that charges user credit card. As shown by the policy, the purpose of the data collection is for

```
<STATEMENT>
 <CONSEQUENCE>
   We charge your credit card for your purchase order.
   Information is destroyed once transaction complete.
 </CONSEQUENCE>
 <PURPOSE><sales/></PURPOSE>
 <RECIPIENT><ours/></RECIPIENT>
 <RETENTION><stated-purpose/></RETENTION>
 <DATA-GROUP>
  <DATA ref="#user.payment.creditcard">
    <category> <purchase /> </category>
  </DATA>
 </DATA-GROUP>
</STATEMENT>
```

Fig. 1. Sample P3P Statement

`sales`. The information will be maintained at the website (as specified by `ours` in the `RECIPIENT` element), for a limited time until the transaction is completed (as indicated by `stated-purpose`).

P3P provides many predefined constants for each element of a statement. For instance, the following are several typical values for the `PURPOSE` element: (1) `current`: for the current one-time activity, (2) `admin`: for website administration, and (3) `telemarketing`: the information can be reused for promotion of a product later. For another example, the value of the `RECIPIENT` element can be, e.g., `ours` (the website owner), `delivery` (the delivery service), `same` (including other collaborators performing one-time use of the information), and `public`.

Clearly, a P3P policy is an access control specification that describes how information is distributed, stored, and destroyed. Many consider the policy enforcement as a requirement engineering problem [15]. We are interested in automated verification and auditing of policy enforcement, using a logic based framework. This requires a simplified formal model which avoids semantics problems in P3P.

3 PV Framework

The PV logic framework intends to model the information flow among entities of a web application, including software components as well as stakeholders. We assume an infinite model in this section, but later in verification, Alloy Analyzer works on a finite model only.

3.1 Data Model

Let $\mathcal{E}$ be an infinite but countable set of *entities* in a web application. An entity represents an *atomic* real entity of the world. It can be a person, a database, and an organization. Let $\mathcal{D}$ be an infinite and countable set of *data items*. Each data item $d \in \mathcal{D}$ is an atomic piece of information (e.g., the name of a person, a credit card number, etc.). The data model is flattened, i.e., we do not allow hierarchical data structure in the model like [7]. Data typing is defined using set containment as in relational logic [18]. A *data type* is a set of data items. A data type D_1 is a *subtype* of D_2 iff $D_1 \subseteq D_2$. If a data item $d \in D$, we say that d has the type D.

3.2 Web Application

A web application is modeled as a reactive software system, consisting of a finite collection of *servlets*. This corresponds to many existing web application platforms such as PHP, JavaEE, and ASP.Net. A *servlet* is a function which takes an HTTP request as input, and returns an HTTP response as output. It may have side effects, e.g., manipulating backend database, and sending emails. Very often the business organization (owner of the website) may have routinely performed procedures (e.g., clearing customer database monthly etc.). They are similar to servlets in that they have side effects. We generalize the notion to *actions* for capturing the semantics of both servlets and business procedures.

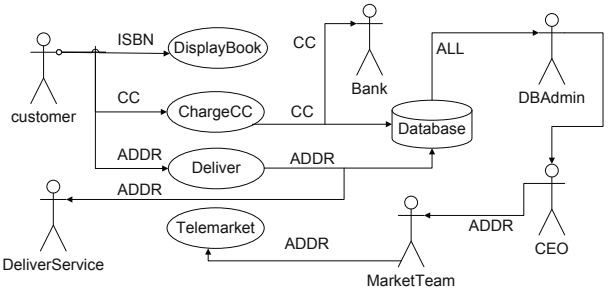


Fig. 2. Sample Bookstore Application

Definition 1. *A web application W is a tuple $(\mathcal{D}, \mathcal{E}_w, \mathcal{A}, \mathcal{P}, \mathcal{E}_e, P)$ where $\mathcal{D}$ is a a set of data items, $\mathcal{E}_w$ is a finite set of entities in W, $\mathcal{A} \subseteq \mathcal{E}_w$ is a finite set of actions (including servlets and business procedures), $\mathcal{P}$ is a finite set of purposes, and $\mathcal{E}_e$ is a set of entities in the environment. $P : \mathcal{A} \rightarrow 2^{\mathcal{P}}$ associates a set of purposes to each action.*

Example 1. Shown in Figure 2 is the architecture of a bookstore web application. $\mathcal{E}_w$ ={`DisplayBooks`, `ChargeCC`, `Deliver`, `Telemarket`, `DB`, `DBAdmin`, `MarketTeam`, `CEO`} is the set of entities within the application. $\mathcal{A}$ contains the first four elements (i.e., the servlets) of $\mathcal{E}_w$. $\mathcal{E}_e$ = {`Customer`, `Bank`, `DeliveryService`} contains three external entities that interact with the web application.

$\mathcal{P}$ has three elements: `purchase`, `delivery`, and `marketing`. Clearly, for the three servlets, their purposes can be defined as below. `P`(`ChargeCC`) = {`purchase`, `delivery`}, `P`(`Deliver`) = {`delivery`}, and `P`(`Telemarket`) = {`marketing`}.

$\mathcal{D}$ consists of data items that belong to three data types: `CC` (credit card number), `ADDR` (address), and `ISBN` (book id). In the rest of the paper, we use the above example as a motivating case study.

3.3 Action

We assume that each servlet (action) will eventually complete, i.e., it is never trapped in an infinite loop. In practice, this is guaranteed by the time-out action of web server. An action is atomic. Formally, an action is defined as a transition rule that manipulates predicates on the "knowledge" of entities.

An *action* is a first order relational logic formula [18], built on a predicate named `know`. When a predicate is primed, it represents the value of the predicate in the next system state of a web application. All free variables in the formula are regarded as input parameters.

Example 2. We list the specification of all servlets in Example 1, which can be generated by the static analysis algorithm in §5.

1. `DisplayBooks`: $\forall x \in \mathcal{E}_w \cup \mathcal{E}_e \ \forall d \in \mathcal{D}$: `know`(x, d) = `know`$'(x, d)$.
   The servlet does not have any side effects. Thus it does not change the valuation of predicate `know`.

2. `ChargeCC`: $cc \in \texttt{CC} \wedge \texttt{know}'(\texttt{DB}, cc) \wedge \texttt{know}'(\texttt{Bank}, cc) \wedge \forall x \in \mathcal{E}_w - \{\texttt{DB}, \texttt{Bank}\}\ \forall d \in \mathcal{D} : \texttt{know}(x, d) = \texttt{know}'(x, d) \wedge \forall d \in \mathcal{D} - \{cc\} : \texttt{know}(\texttt{Bank}, d) = \texttt{know}'(\texttt{Bank}, d) \wedge \texttt{know}(\texttt{DB}, d) = \texttt{know}'(\texttt{DB}, d)$
Note that cc (the free variable) is the input parameter of the servlet. `ChargeCC` saves the information to local database and submits the information to `Bank`. Here $\texttt{CC}, \texttt{DB}, \texttt{Bank}$ are all constants. The last two clauses (with $\forall$ quantifiers) keep the valuation of the predicate for all other entities and data. In the rest of the paper, we use $\texttt{same\_except}((\texttt{DB}, cc), (\texttt{Bank}, cc))$ to represent such an assignment that maintains predicate valuations in the new system state, except for pairs $\{(\texttt{DB}, cc), (\texttt{Bank}, cc)\}$.
3. `Deliver`: $cc \in \texttt{CC} \wedge ad \in \texttt{ADDR} \wedge \texttt{know}'(\texttt{DeliveryService}, ad) \wedge \texttt{know}'(\texttt{DeliveryService}, cc) \wedge \neg\texttt{know}'(\texttt{DB}, cc) \wedge \texttt{same\_except}(\{(\texttt{DB}, cc), (\texttt{DeliveryService}, \{cc, ad\})\})$.
The `Delivery` servlet is similar to `ChargeCC`. The difference is that the record cc is now removed from `DB`, to achieve the `stated-purchase` property, i.e., the credit card information is destroyed after the transaction is completed.
4. `Telemarket`: it is the same as `DisplayBooks` and does not have any side effects (it simply takes a list of email addresses and sends out emails).

There are certain cases that the information flow between components may not be extracted by a static analysis. We use the notion of `Pipe` to specify such flow directly.

Definition 2. *Let $D \subseteq \mathcal{D}$ be a data type and e_1 and e_2 two entities in $\mathcal{E}_w \cup \mathcal{E}_e$. We say $\texttt{Pipe}(e_1, e_2, D)$ if information of type D flows from e_1 to e_2.*

Each $\texttt{Pipe}(e_1, e_2, D)$ can be translated into a transition rule

$$\forall d \in D : \texttt{know}(e_1, d) \Rightarrow \texttt{know}'(e_2, d)$$

Example 3. The information flow in Figure 2 can be represented using the conjunction of three pipes: (1) $\texttt{Pipe}(\texttt{DB}, \texttt{DBAdmin}, \mathcal{D})$, (2) $\texttt{Pipe}(\texttt{DBAdmin}, \texttt{CEO}, \mathcal{D})$, and (3) $\texttt{Pipe}(\texttt{CEO}, \texttt{MarketTeam}, \texttt{ADDR})$.

3.4 Modeling Security Policy

A privacy policy is about the access control of information: (1) if the information is used for intended use, (2) if the information is destroyed after the specified retention, and (3) if the information is only known by a restricted group of people. This is reflected by the `PURPOSE`, `RETENTION`, and `RECIPIENT` elements in a P3P policy.

Natural language specification, however, lacks formal semantics and can often cause confusion. Consider, for example, the term `stated-purpose` in P3P specification ("`Information is retained to meet the stated purpose. This requires information to be discarded at the earliest time possible`" [30]). The "earliest time" can be interpreted in many ways, e.g., when the transaction is completed or when the website owner feels no needs of the data.

Temporal logic can be used to nicely capture privacy policies. We assume that readers are familiar with computation tree logic (CTL) [10]. In the following we define CTL-FO, an extension of CTL which allows free mixture of first order quantifiers. The definition is borrowed directly from [11,13], with slight modification for relational logic.

Definition 3. *Let $\mathcal{D}$ be a data domain and $\mathcal{P}$ be a finite set of predicates. The CTL-FO formulas are defined as below:*

1. *Let $p \in \mathcal{P}$ be a predicate with arity n, and $\boldsymbol{x}$ be a vector of variables and constants ($|\boldsymbol{x}| = n$). Then $p(\boldsymbol{x})$ is a CTL-FO formula.*
2. *If f and g are CTL-FO formulas, then all of the following are CTL-FO formulas: $f \wedge g$, $\neg f$, $f \vee g$, $\mathbf{AX}\ f$, $\mathbf{EX}\ f$, $\mathbf{AF}\ f$, $\mathbf{EF}\ f$, $\mathbf{A}\ f\ \mathbf{U}\ g$, $\mathbf{E}\ f\ \mathbf{U}\ g$, and $\mathbf{A}\ f\ \mathbf{R}\ g$, and $\mathbf{E}\ f\ \mathbf{R}\ g$. Here $\mathbf{A}$ (universal path quantifier), $\mathbf{E}$ (existential path quantifier), $\mathbf{X}$ (next state), $\mathbf{F}$ (eventually), $\mathbf{G}$ (globally), $\mathbf{U}$ (until), and $\mathbf{R}$ (release) are standard CTL temporal operators.*
3. *If f is a CTL-FO formula and $D \subseteq \mathcal{D}$ is a data type, then $\forall x \in D : f$ and $\exists x \in D : f$ are CTL-FO formulas.*

Definition 4. *A CTL-FO formula φ is said to be* well formed *if φ has no free variables, and every quantified variable v appears in some predicate in φ.*

The semantics of CTL-FO formula can be defined by directly extending the CTL semantics in [10] with the addition of the following two semantics rules. Given a formula φ and a free variable v in φ, $\varphi_{x \leftarrow d}$ is the result of replacing every x with constant d. Then given a Kripke structure M and a state s:

1. $M, s \models \forall x \in D : \varphi \ \Leftrightarrow$ for each value d in D: $M, s \models \varphi_{x \leftarrow d}$
2. $M, s \models \exists x \in D : \varphi \ \Leftrightarrow$ there exists a value d in D s.t. $M, s \models \varphi_{x \leftarrow d}$

Note that in the above definition, the $\forall x \in D : \varphi$ and $\exists x \in D : \varphi$ are required to be well-formed. In another word, $\varphi_{x \leftarrow d}$ has no free variables. In the following we introduce one security policy specified using CTL-FO for Example 1.

Example 4. Policy 1: any credit number collected by the `ChargeCC` servlet is eventually destroyed by the web application, i.e., no entities in the bookstore web application knows about the credit card number eventually. The property can be expressed as below:

$$\forall d \in \mathtt{CC} : \mathbf{AG}(\mathtt{know}(\mathtt{DB}, d) \ \Rightarrow \ \mathbf{AF}(\forall x \in \mathcal{E}_w : \neg \mathtt{know}(x, d)))$$

3.5 Conformance Verification Problem

Given a web application $W = (\mathcal{D}, \mathcal{E}_w, \mathcal{A}, \mathcal{P}, \mathcal{E}_e, P)$, it is straightforward to define a Kripke structure on W, written as $M(W)$. The basic idea is that each state of $M(W)$ is a distinct valuation of predicate `know` on each pair of entity (in $\mathcal{E}_w \cup \mathcal{E}_e$) and data item (in $\mathcal{D}$). The initial state s_0 of W needs to be manually defined by the designer. Transitions between states can be derived by the action rules of $\mathcal{A}$. Then the conformance verification problem is defined as the following.

Definition 5. *Let W be a web application, and (M, s_0) the derived Kripke structure and the initial state. W conforms to a CTL-FO formula φ iff $M, s_0 \models \varphi$.*

4 Verification

4.1 Overview

This section introduces a symbolic verification paradigm which takes a web application specification and a CTL-FO formula as input. It either outputs "yes", or generates a firing sequence of servlets (or other actions) that leads to the violation of the property. We rely on the Alloy Analyzer [17] for model checking if a PV model satisfies a privacy policy specified in CTL-FO logic. Using SAT-based model finder Kodkod [28], Alloy performs scope-restricted model finding. The Alloy specification supports first order relational logic [18], which is very convenient for specifying the transition system of a PV model.

The verification paradigm consists of the following steps:

1. *Translation from PV Transition System to Alloy:* It consists of two parts: (a) specification of all PV data entities using the Alloy type system, and (b) translation of each action into a first order relational logic formula that contains both current/next state predicates. Notice that Alloy is originally designed for static model analysis, the state of a PV transition system has to be explicitly modeled to simulate a Kripke structure. Subsection 4.2 presents the technical details about this part of translation.
2. *Translation from a CTL-FO formula to Alloy predicates and assertions:* Alloy itself does not support temporal logic. This translation is about simulating the fixpoint computation of temporal operators. Subsection 4.3 presents the details.
3. *Verification using Alloy:* Given a property (expressed as Alloy assertions), Alloy is able to find a model that violates the assertion using a finite model/scope search. The error trace can be easily identified from the visual representation provided by Alloy. In the visual model, an error trace comprises of a sequence of PV states. Note that these states and their transition relation are already explicitly encoded in the model.

4.2 Translating PV Transition System to Alloy

The translation algorithm is straightforward. Currently, for the case study example, the translation is accomplished using manual simulation of the algorithm. In our future work, the translation will be automated.

Taking Example 1 as an example, its Alloy specification is given in Figure 3. The specification contains three parts: (1) the general data schema that defines the world of entities and actions; (2) the specific data setting related to the web application, e.g., its actions and stakeholders, (3) the formal definition of actions, where each action (servlet) is modeled as a parametrized transition rule.

The World Schema: The first section of the Alloy specification defines the general data schema for all PV model specifications in Alloy. Here `Object` is a generic type, which has three subtypes: `WA` (all entities within the web application), `Env` (all entities of the environment), and `Data` (all data items). The

```
module bookstore

//1. World Schema
abstract sig Object {}
abstract sig WA, Env, Data extends Object {}
abstract sig Actions,  Entities extends WA{}
abstract sig actionStatus{}
one sig RUN, SLEEP, READY extends actionStatus{}
abstract sig Purpose {}
sig State{
  know: (WA +Env) -> Data,
  prev: one State,
  actstate: Actions -> actionStatus
}{
  all x: Actions | some status: actionStatus |
      x -> status in actstate
}
sig initState extends State {}
fact generalInitState{
  all x: initState |
    (x.prev = x and
       all y: Actions | x.actstate[y] = SLEEP
    ) and
    (some y: State -initState | x in y.^prev)
}
fact factAllStates{
  all x: State - initState | some y : initState |
      y in x.^prev
}
fact TransitionRelation{
  all x: State   | all y: State - initState |
      ( (x in y.prev => Transition[x,y]) and
      (Transition[x,y] => x in y.prev) )
}

//2. Web Application Specific Setting (Bookstore)
sig NAME, CC, ADDR, ID extends Data{}
one sig DisplayBooks, ChargeCC,
    Deliver, Telemarket extends Actions {}
one sig DB, DBAdmin, MarketTeam,
    CEO extends Entities {}
one sig Bank, DeliverService, User extends Env {}
...

//3. Actions (Servlets and Pipes)
...
pred pChargeCC [s,s': State, d: CC]{
   ChargeCC->READY in s.actstate and
   (
    s'.know = s.know + {DB->d} +{Bank->d} &&
    s'.prev = s &&
    s'.actstate = s.actstate - {ChargeCC->READY}
      + {ChargeCC->SLEEP} - {Deliver->SLEEP}
      + {Deliver->READY}
   )
}

pred pipe[s,s': State, e1, e2: Object, D:  Data]{
  (some d: D | e1->d in s.know
     && !(e2->d in s.know))
  &&(
     s.know in s'.know &&
     (all x: (WA+Env) | all y:Data |
          (x->y in s'.know-s.know) <=>
          (x=e2 && e1->y in s.know
           && !(e2->y in s.know)) )
     && s'.prev=s
     && s'.actstate = s.actstate
  )
}

pred customer[s,s':State]{
  (DisplayBooks->SLEEP in s.actstate and
          s'.actstate = s.actstate
          - {DisplayBooks->SLEEP}
          + {DisplayBooks->READY}
    &&  s'.prev=s  && s'.know=s.know) or
   ...
}

pred Transition[s,s':State]{
  //servlets
  pDisplayBooks[s,s'] or
  pTelemarket[s,s'] or
  (some d:CC | pChargeCC[s,s',d]) or
  (some d:ADDR | some d2:CC |
       pDelivery[s,s',d,d2]) or
  //pipes
  pipe[s,s',DB,DBAdmin,Data] or
  pipe[s,s',DBAdmin,CEO,Data] or
  pipe[s,s',CEO,MarketTeam,ADDR] or
  //other actions
  customer[s,s']

}


//4. CTL-FO to predicates and assertion
pred ef[s:State,d:Data]{
    some s': State | (CEO ->d in s'.know)
         && s in s'.*prev
}

pred fa[s:State]{
  all d: Data | (DB->d in s.know) => ef[s,d]
}

assert AGProperty{
  all s: State | fa[s]
}
```

Fig. 3. Sample ALLOY Specification

`WA` entities consist of `Actions` (like servlets) and `Entities` (like databases). To simulate each action as a transition, we define three constants to denote status of an action: `RUN`, `READY`, and `SLEEP` (the `RUN` status is actually not needed as each transition rule is atomic). To build a Kripke structure of the transition system, we declare `State` which includes `Knows` (tracking the knowledge of each entity on data items), `prev` (the previous state), and the status of each action. The restriction "`all x: Actions | some status: actionStatus | x -> status in actstate`" requires that all action has a status. We also declare that there is one or more initial states (as described by `fact generalInitState`): the status of all actions are set to `SLEEP`. More details of the initial states are defined in

the section on web application specific settings. Note that Alloy specification is declarative, thus the order of Alloy statements does not affect the semantics. Finally, the fact `factAllStates` requires that all instances of `State` enumerated by Alloy should be contained in the transitive closure of `prev` link to an initial state (note `^` is the non-reflexive transitive closure operator).

Web Application Specific Settings: The settings related to the Bookstore application are specified, e.g., the subtypes `NAME`, `CC`, and `ADDR`, the servlets, and external entities.

Actions (Servlets, Pipes): Each action is modeled as a predicate in Alloy. A typical example is `pChargeCC` which represents the transition rule for the `ChargeCC` servlet. The predicate takes three parameters: `s` and `s'` are the current and next states. `d` is a credit card number. Clearly, the predicate specifies that in the next state, the `DB` is able to know `d`, and it updates the `prev` link and the action status correspondingly. The `pipe` predicate models a template of piping information. Given entities e_1 and e_2, if e_1 knows d, then in the next state, e_2 also knows d. Notice that there is a `customer` action, which non-deterministically invokes `DisplayBooks` and `ChargeCC`.

The `Transition` predicate defines the transition relation, which is composed of the predicates that model the servlets, pipes, and other actions/roles in the system. Clearly, with `Transition` and `factAllStates`, a Kripke structure is formed using the `prev` field of each state. Alloy will analyze the states reachable from the initial state only.

4.3 Translating CTL-FO Formula

A CTL-FO Formula can be translated into one Alloy assertion and a collection of Alloy predicates. The translation runs from top to bottom, following the syntax tree. The top level CTL-FO formula is formulated as an assertion. Then each component is modeled as an Alloy predicate (with one input parameter on PV state, and the necessary parameters for quantified variables). The fixpoint computation of CTL formula can be modeled using first order logic plus the `prev` link (which models the transition relation between states).[1]

Take the following CTL-FO formula as one example.

$$\mathbf{AG}(\forall d \in \texttt{Data} : \texttt{know}(\texttt{DB}, d) \;\Rightarrow\; \mathbf{EF}(\texttt{know}(\texttt{CEO}, d)))$$

The top level **AG** property is first translated into an Alloy assertion (as shown in `AGProperty`). The **EF** formula is defined as a predicate `ef` which has two parameters: `s` and `d`. Here `d` is a variable which is restricted by the universal quantifier, and `s` is a PV state. The predicate `ef[s,d]` is true iff at `s` and for data item `d` eventually there is a path leads to a state that `CEO` knows `d`. This is defined using the formula inside `ef`, which leverages the Kleene closure of the `prev` link (see "`s in s'.*prev`").

[1] Currently, only the least fixed point operators are handled.

4.4 Initial Experimental Results

We performed an initial experiment with the Alloy model. The verification cost explodes quickly with the number of states. The system runs out of memory when it exceeds 50 PV states, on a PC with 4GM RAM.

5 Static Information Flow Analysis

This section proposes a semi-automatic static analysis algorithm that produces the PV model. The algorithm has not been implemented, but the idea is straightforward. The algorithm consists of four stages: (1) a static code analysis that extracts the external entities, e.g., file system and databases that are accessed by servlets, (2) a manual definition stage, where the designers supply the rest of the roles and stakeholders, e.g., the `MarketTeam` and `CEO` in Example 1, (3) modeling of all system calls, e.g., to define the data sink and operations performed by system calls such as JDBC `executeUpdate`, and (4) a fully-automatic analysis of servlet bytecode, which extracts information flow and builds the transition system. We omit the details of steps (1) to (3), and concentrate on step (4).

5.1 Path Transducer

The fully automatic static code analysis assumes that each web servlet can be represented by a set of *path transducers*. A *path transducer* of a servlet is essentially one possible execution path of the servlet from its entry to exit (by unwrapping loops, branches, and tracing into function calls). The analysis can be inter-procedural in the sense that it traces into every function defined by the web application, but not into the function body of any system functions provided by the environment (e.g., OS system calls).

Definition 6. *A* path transducer T *is a tuple* $(\boldsymbol{I}, \boldsymbol{M}, \boldsymbol{S})$ *where* $\boldsymbol{I}$ *is a finite set of input parameters,* $\boldsymbol{M}$ *is a finite set of variables, and* $\boldsymbol{S}$ *is a sequence of statements. Each* statement *has one of the following two forms (letting* $v \in \boldsymbol{M}$*,* $V \subseteq \boldsymbol{M}$*, and* C *be a sequence of constants):*

1. (Assignment) $v := E(V, C)$.
2. (System Call) $v := f(V, C)$.

Here $\boldsymbol{M}$ includes all static, stack, and heap variables that could occur during the execution, as unwrapping is bounded. E is an arithmetic or logical expression on V and C. f has to be a *system call* that is provided by the external environment of the web application.

Example 5. Figure 4 presents a simple Java servlet that adds a user to a web email system. It calls a self-defined `massage()` function to sanitize user input. Then it submits an `INSERT` query to the database. The statement sequence of a sample path transducer is given in Figure 5. Here system calls are replaced by a shorter name for simplicity (e.g., `response.getWriter(...)` is replaced

```
1 protected void processRequest(
2  HttpServletRequest request ...){
3  PrintWriter out = response.getWriter();
4  String sUname = request.getParam("sUname");
5  String sPwd = request.getParameter("sPwd");
6  Connection conn = DM.getConnection("...");
7  Statement stmt = conn.createStatement();
8  String strCmd= "INSERT ..."
9     + massage(sUname) + ... + massage(sPwd);
10 int n = stmt.executeUpdate(strCmd);
11 if(n>0) out.println("Welcome"+sUname);
12 }
13 protected String massage(String str){
14   return str.replaceAll("'", "''");
15 }
```

Fig. 4. Add Member Servlet

```
1  out = f1();
2  sUname = input_sUname;
3  sPwd = input_sPwd;
4  //new entity DBConn_Addr
5  stmt = f2();
6  //now call massage(sUname)
7  str = sUname;
8  s1 = f3(str,"'", "''");
9  //now call massage(sPwd)
10 str = sPwd;
11 s2 = f3(str,"'", "''");
12 //now call executeUpdate
13 strCmd = f4("INSERT..."+s1+...+s2);
14 n = f5(stmt, strCmd)
```

Fig. 5. Sample Path Transducer

by `f1(...)`). All branch statements and calls of self-defined functions (i.e., `massage`) are removed by unwrapping. For example, `f3` (`String.Replace`) is called twice because the execution enters the `massage` function twice. Temporary variables, e.g., `s1` and `s2`, are created to handle the temporary function call results. But only a finite number of them are needed because the unwrapping depth is bounded. By defining the data sinks of system calls and applying string analysis such as [31], it is possible to extract flow information from system calls,e.g., from `f5` (the `executeUpdate`) function.

Symbolic execution [22] can be used to extract path transducers from the bytecode of servlets. A typical approach is to instrument the virtual machine and skip the real decision of a branch statement so that both branches can be covered (see e.g., [3]). An alternative approach is to instrument the bytecode of the web application being inspected, to change its control flow, using tools such as Javassist [9]. A servlet may have an infinite number of path transducers.

5.2 Static Analysis for Constructing Transition System

Figure 6 displays the static analysis algorithm. It takes two inputs: a path transducer $\mathcal{T}$ and a mapping ρ that associates each input variable with the corresponding private information. `CalcInfoFlow` returns a collection of tuples that

```
1  Procedure CalcInfoFlow(T = (M, I, S), ρ : I → 2^D)
2  //T is a path transducer, ρ is a mapping from input variables to the data items.
3     know := ρ
4     foreach s in S do:
5       case M_i := E(V, C):
6         for each v ∈ V: for each x s.t. (v, x) ∈ know: know.add(M_i,x)
7       case M_i := f(V, C):
8         for each v ∈ V: for each x s.t. (v, x) ∈ know: know.add(M_i,x)
9         Let e be the data sink of f
10        for each x s.t. (v, x) ∈ know:
11          know.add(e, x)
12    return {(x, d) | (x, d) ∈ know ∧ x ∉ M ∪ I}
```

Fig. 6. Static Analysis Algorithm

represent the new facts about the knowledge of information by entities. `know` models the knowledge of all entities (including variables) on private information. We populate its contents from the initial knowledge of ρ. Whenever an assignment is reached, the knowledge of all variables on the right will also be the knowledge of the left hand side. When a statement is an invocation of system call, the information is propagated to the proper data sink. Finally, the collection of `know` tuples is returned. Then we can easily generate the transition rule that models the servlet. The following lemma implies that we do not need an infinite collection of path transducers to compute the complete result.

Lemma 1. *For any servlet a_i (letting ρ be the mapping in Figure 6), there exists a* finite *set of path transducers for a_i (letting it be Ψ) s.t. for any path transducer set for a_i (letting it be Φ) the following is true:*

$$\bigcup_{\tau \in \Phi} \texttt{CalcInfoFlow}(\tau, \rho) \subseteq \bigcup_{\tau \in \Psi} \texttt{CalcInfoFlow}(\tau, \rho)$$

6 Related Work

Using formal methods to model privacy has long been an area of interest (see [29] for a comprehensive review). One typical application is the study of the semantics of security policies. While P3P [30] is able to express a security policy in a machine understandable format (i.e., XML), its lack of formal semantics is often under debate as well as criticism. Barth and Mitchell pointed many pitfalls of P3P and EPAL in [7], e.g., the early termination causing non-robustness, the lack of distinction between provider and consumer perspectives, and the missing of fine-grained access control on subset of data groups. Similarly, Yu *et al.* showed that a P3P privacy policy can have multiple statements conflicting with each other [32] (e.g., imposing multiple retention restrictions over one data item). There are several proposals to fix the problems of P3P, e.g., the EPAL standard [26], and the privacy policy based on semantic language DAML-S [19]. This paper uses a first order extension of computation tree logic (CTL) for modeling privacy policies. The benefit of using temporal logic is the simplicity of model and the very expressive temporal operators for expressing the notions of information control that is related to time. The idea of using temporal logic for specifying privacy policies is not new. In [6], Barth *et al.* presented an extension of Linear time Logic (LTL) for modeling a variety of privacy policy languages. Similar efforts include the `REVERSE` working group on trust and policy definition [25], led by M. Baldoni. Compared with [6], our contribution is the modeling of a web application as a transition system and the verification scheme that addresses the model checking of first order temporal logic, which is not discussed in [6].

Deployment of privacy policies (e.g., [1]) is not the concern of this paper. We are more interested in the *enforcement* (or conformance check) of privacy policies. Many works enforce privacy policies using a *top-down* fashion, e.g., role engineering in the software architectural design stage (assigning permission rights of storing and distributing information to stakeholders) [15], enforcing

P3P policy in a web application by leveraging existing enterprise IT systems [4], asking designers to follow specific design patterns (IBM Declarative Data Privacy Monitoring) [16], and enforcing data access control using JIF (a variant of Java) [14]. This work adopts a *bottom-up* approach: given an existing web application, we perform static analysis on its bytecode, extract a formal model on information flow, and verify if the model satisfies a privacy policy. The conservative code analysis helps to increase confidence in privacy protection.

This work follows the general methodology of symbolic model checking and its applications to web services [8,12]. However, we face the challenge of handling first order logic, which in general is an undecidable problem. Our approach is to bound the scope of the model, and rely on Alloy Analyzer [17] to perform a bounded model checking. Alloy has been widely applied to many interesting problems, e.g., multicast key management [27], correcting naming architecture [21], and solving relational database constraints [20]. Most of its applications are applied to static models, while we attempted the modeling of a dynamic transition system (and computing fixpoint of first order temporal logic formula) using Alloy. The experiment shows that the verification cost explodes quickly with the number of states. More efficient verification can be performed, e.g., by invoking the Kodkod model finder [28] directly. An alternative is to prove privacy preservation by studying refinement relation between transition systems [2].

7 Conclusion

This paper has presented a logic based framework for reasoning about the privacy protection provided by a web application. A first order extension of the computation tree logic is used to specify a privacy policy. Then a formal transition model is constructed by performing a semi-automatic code level analysis of the web application. The verification relies on Alloy Analyzer and is performed on a finite model, expressed in first order relational logic. Our future directions include implementing the PV framework, applying it to non-trivial web applications, and exploring more efficient constraint solving techniques.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Implementing p3p using database technology. In: Proceedings of 19'th International Conference on Data Engineering, pp. 595–606 (2003)
2. Alur, R., Černý, P., Zdancewic, S.: Preserving secrecy under refinement. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 107–118. Springer, Heidelberg (2006)
3. Anand, S., Păsăreanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to java pathfinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
4. Ashley, P.: Enforcement of a p3p privacy policy. In: Proceedings of the 2nd Australian Information Security Management Conference, pp. 11–26 (2004)

5. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: Verifying the Conformance of Web Services to Global Interaction Protocols: A first step. In: Bravetti, M., Kloul, L., Tennenholtz, M. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 257–271. Springer, Heidelberg (2005)
6. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: Framework and applications. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 184–198 (2006)
7. Barth, A., Mitchell, J.C.: Enterprise privacy promises and enforcement. In: Proceedings of the 2005 Workshop on Issues in the Theory of Security, pp. 58–66 (2005)
8. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: IEEE Symposium on Logic in Computer Science, pp. 428–439 (1990)
9. Chiba, S.: Getting started with javassit, http://www.csg.is.titech.ac.jp/~chiba/javassist/html/index.html
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
11. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web services. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2004, pp. 71–82 (2004)
12. Fu, X., Bultan, T., Su, J.: Model checking XML manipulating software. In: Proceedings of the 2004 Int. Symp. on Software Testing and Analysis (ISSTA), pp. 252–262 (2004)
13. Hallé, S., Villemaire, R., Cherkaoui, O., Tremblay, J., Ghandour, B.: Extending model checking to data-aware temporal properties of web services. In: Proceedings of 2007 Web Services and Formal Methods, 4th International Workshop, pp. 31–45 (2007)
14. Hayati, K., Abadi, M.: Language-based enforcement of privacy policies. In: Proceedings of Privacy Enhancing Technologies Workshop, pp. 302–313 (2005)
15. He, Q., Anton, A.: A framework for modeling privacy requirements in role engineering. In: Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 1–15 (2003)
16. IBM. Declarative Data Privacy Monitoring, http://www.redbooks.ibm.com/redbooks/pdfs/sg246999.pdf
17. Jackson, D.: Alloy 3 Reference Manual, http://alloy.mit.edu/reference-manual.pdf
18. Jackson, D.: Automating first-order relational logic. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 130–139 (2000)
19. Kagal, L., Paolucci, M., Srinivasan, N., Denker, G., Finin, T., Sycara, K.: Authorization and privacy for semantic web services. IEEE Intelligent Systems 19(4), 50–56 (2004)
20. Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, S.: Query-aware test generation using a relational constraint solver. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 238–247 (2008)
21. Khurshid, S., Jackson, D.: Correcting a naming architecture using lightweight constraint analysis. Technical report, MIT Lab for Computer Science (December 1998)
22. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

23. House of Representatives. Health insurance portability and accountability act of (1996), http://www.gpo.gov/fdsys/pkg/CRPT-104hrpt736/pdf/CRPT-104hrpt736.pdf
24. Rajamani, S.K., Rehof, J.: Conformance checking for models of asynchronous message passing software. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 166–179. Springer, Heidelberg (2002)
25. REVERSE - policies & trust, http://cs.na.infn.it/rewerse/pubs.html
26. W3C Member Submission. Enterprise privacy authorization language (epal 1.2.), http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/
27. Taghdiri, M., Jackson, D.: A lightweight formal analysis of a multicast key management scheme. In: Proceedings of Formal Techniques of Networked and Distributed Systems, pp. 240–256. Springer, Heidelberg (2003)
28. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, pp. 632–647 (2007)
29. Tschantz, M.C., Wing, J.M.: Formal methods for privacy. In: Proceedings of the 2nd World Congress on Formal Methods, pp. 1–15 (2009)
30. W3C. Platform for privacy preferences (p3p), http://www.w3.org/P3P/
31. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 32–41 (2007)
32. Yu, T., Li, N., Antón, A.I.: A formal semantics for p3p. In: Proceedings of the 2004 Workshop on Secure Web Service, pp. 1–8 (2004)

Generalised Computation of Behavioural Profiles Based on Petri-Net Unfoldings

Matthias Weidlich, Felix Elliger, and Mathias Weske

Hasso Plattner Institute at the University of Potsdam, Germany
{Matthias.Weidlich,Mathias.Weske}@hpi.uni-potsdam.de,
Felix.Elliger@student.hpi.uni-potsdam.de

Abstract. Behavioural profiles have been proposed as a concept to judge on the behavioural consistency of process models that depict different perspectives of a process. These profiles describe the observable relations between the activities of a process model. Consistency criteria based on behavioural profiles are less sensitive to model projections than common equivalence criteria, such as trace equivalence. Existing algorithms derive those profiles for unlabelled sound free-choice workflow nets efficiently. In this paper, we generalise the computation of behavioural profiles by relaxing the aforementioned assumptions. First, we introduce an algorithm that derives behavioural profiles from the complete prefix unfolding of a bounded Petri net. Hence, it is applicable in a more general case. Second, we lift the concept to the level of labelled Petri nets. We also elaborate on findings of applying our approach to a collection of industry models.

1 Introduction

Notions of behavioural similarity and consistency of business process models have a broad field of application, e.g., for model retrieval [1] or management of process variants [2]. Targeting at the analysis of behavioural consistency between business process models used as a specification and workflow models representing the implementation of the process, behavioural profiles have been proposed as a basis for such consistency analysis [3]. These profiles capture the relations between pairs of activities in terms of their order of potential execution.

Behavioural profiles have been introduced for unlabelled Petri nets along with efficient algorithms for their computation. For sound free-choice Petri nets that have dedicated start and end places (aka workflow nets), the profile is computed in $O(n^3)$ time with n being the number of nodes of the net [3]. Under certain structural assumptions, computation is even more efficient [4]. In this paper, we generalise the computation of behavioural profiles. That is, we introduce an approach for their computation that imposes solely one restriction on a Petri net system — the system has to be bounded. We derive the behavioural profile from the complete prefix unfolding of a Petri net system. We also report on findings of applying our approach to a collection of industry models. In addition, we show how the notion of a behavioural profile is lifted to the level of labelled nets.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 101–115, 2011.

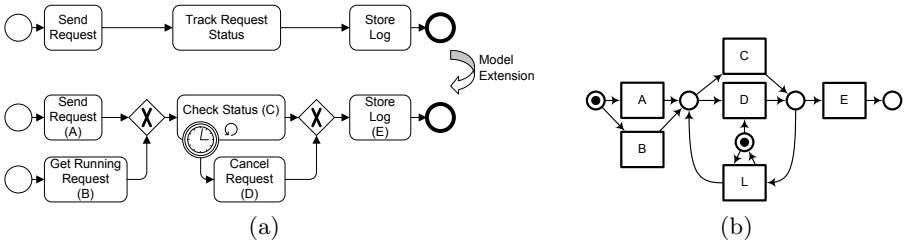


Fig. 1. Process models in BPMN (a); one of the corresponding net systems (b)

Due to their focus on order dependencies, consistency measurement based on behavioural profiles is not affected by model extensions (or model projections, respectively) that impact on the causal dependencies between activities. For instance, Fig. 1(a) depicts two process models in BPMN, the lower one being an extended version of the upper one. A new action to start the process, i.e., *get running request (B)*, has been introduced in the course of model refinement. Apparently, this activity breaks the causal dependency that can be observed between activities *send request* and *store log* in the upper model. Nevertheless, the behavioural profile states that both activities show the same order of potential execution. There are further areas of application for behavioural profiles. They can be used to quantify behavioural deviation between two process models [3], to measure the compliance of process logs [5], and enable change propagation [6].

As mentioned before, there are efficient algorithms for the computation of behavioural profiles that impose the following restrictions.

- The Petri net has to be *sound*. This property is traced back to liveness and boundedness and implies the freedom of deadlocks and livelocks along with the absence of dead transitions. Still, a recent study observed that solely half of the process models in an industry model collection are sound [7]. Moreover, it has been argued that soundness is a rather strict correctness criterion for some use cases. Nevertheless, several weaker correctness criteria, such as weak soundness [8], do not allow for unbounded systems either. Therefore, we argue that boundedness is a reasonable assumption for process models.
- The Petri net has to be *free-choice*. This restriction requires that conflicts and synchronisations of transitions do not interfere. Various constructs of common process modelling languages, such as BPMN, BPEL, and EPCs, can be formalised in free-choice Petri nets [9,10,11]. However, formalisation of exception handling imposes various challenges [12] and typically results in non-free-choice nets (see [9,10]).

The importance of these restrictions is illustrated in Fig. 1(b), which depicts the Petri net formalisation for the lower model in Fig. 1(a). The net is not free-choice due to the modelling of the time-out.

The remainder of this paper is structured as follows. The next section introduces formal preliminaries. Section 3 introduces our approach of deriving the behavioural profile from the complete prefix unfolding along with experimental results. Section 4 defines behavioural profiles for labelled Petri net systems. Finally, Section 5 reviews related work before Section 6 concludes the paper.

2 Background

This section introduces the background of our work. Section 2.1 recalls basic definitions for net systems. Section 2.2 discusses Petri net unfoldings, while Section 2.3 introduces behavioural profiles.

2.1 Net Syntax and Semantics

We recall basic definitions on net syntax and semantics.

Definition 1 (Net Syntax)

- *A* net *is a tuple* $N = (P, T, F)$ *with* P *and* T *as finite disjoint sets of places and transitions, and* $F \subseteq (P \times T) \cup (T \times P)$ *as the flow relation. We write* $X = (P \cup T)$ *for all nodes. The transitive (reflexive) closure of* F *is denoted by* $<$ *(*$\leq$*). A net is* acyclic*, iff* $\leq$ *is a partial order.*
- *For a node* $x \in X$*, the preset is* $\bullet x := \{y \in X \mid (y, x) \in F\}$ *and the postset is* $x\bullet := \{y \in X \mid (x, y) \in F\}$*. For a set of nodes* X'*,* $\bullet X' = \bigcup_{x \in X'} \bullet x$ *and* $X'\bullet = \bigcup_{x \in X'} x\bullet$*.*
- *A tuple* $N' = (P', T', F')$ *is a* subnet *of a net* $N = (P, T, F)$*, if* $P' \subseteq P$*,* $T' \subseteq T$*, and* $F' = F \cap ((P' \times T') \cup (T' \times P'))$*.*

Definition 2 (Net Semantics). *Let* $N = (P, T, F)$ *be a net.*

- $M : P \mapsto \mathbb{N}$ *is a* marking *of* N*,* $\mathbb{M}$ *denotes all markings of* N*.* $M(p)$ *returns the number of* tokens *in place* p*. We also identify a marking* M *with the multiset containing* $M(p)$ *copies of* p *for every* $p \in P$*.*
- *For any two markings* $M, M' \in \mathbb{M}$*,* $M \geq M'$ *if* $\forall\ p \in P\ [\ M(p) \geq M'(p)\ ]$*.*
- *For any transition* $t \in T$ *and any marking* $M \in \mathbb{M}$*,* t *is* enabled *in* M*, denoted by* $(N, M)[t\rangle$*, iff* $\forall\ p \in \bullet t\ [\ M(p) \geq 1\ ]$*.*
- *Marking* M' *is reached from* M *by* firing *of* t*, denoted by* $(N, M)[t\rangle(N, M')$*, such that* $M' = M - \bullet t + t\bullet$*, i.e., one token is taken from each input place of* t *and one token is added to each output place of* t*.*
- *A* firing sequence *of length* $n \in \mathbb{N}$ *is a function* $\sigma : \{0, \ldots, n-1\} \mapsto T$*. For* $\sigma = \{(0, t_x), \ldots, (n-1, t_y)\}$*, we also write* $\sigma = t_0, \ldots, t_{n-1}$*.*
- *For any two markings* $M, M' \in \mathbb{M}$*,* M' *is reachable from* M *in* N*, denoted by* $M' \in [N, M_0\rangle$*, if there exists a firing sequence* σ *leading from* M *to* M'*.*
- *A* net system*, or a* system*, is a pair* (N, M_0)*, where* N *is a net and* M_0 *is the* initial marking *of* N*.*
- *A system* (N, M_0) *is* bounded*, iff the set* $[N, M_0\rangle$ *is finite.*

2.2 Unfoldings of Net Systems

Any analysis of the state space of a net system has to cope with the state explosion problem [13]. Unfoldings and their complete prefixes have been proposed as a technique to address this problem [14,15]. The unfolding of a net system is another, potentially infinite net system, which has a simpler, tree-like structure. We recall definitions for unfoldings based on [16].

Definition 3 (Occurrence Net, Ordering Relations)

- *A pair of nodes* $(x, y) \in (X \times X)$ *of a net* $N = (P, T, F)$ *is in the* conflict relation $\#$*, iff* $\exists\ t_1, t_2 \in T\ [\ t_1 \neq t_2 \wedge \bullet t_1 \cap \bullet t_2 \neq \emptyset \wedge t_1 \leq x \wedge t_2 \leq y\ ]$.
- *A net* $N = (P, T, F)$ *is an* occurrence net*, iff (1)* N *is acyclic, (2)* $\forall\ p \in P\ [\ |\bullet p| \leq 1\ ]$*, and (3) for all* $x \in X$ *it holds* $\neg(x \# x)$ *and the set* $\{y \in X \mid y < x\}$ *is finite. In an occurrence net, transitions are called* events*, while places are called* conditions*.*
- *For an occurrence net* $N = (P, T, F)$*, the relation* $<$ *is the* causality relation*. A pair of nodes* $(x, y) \in (X \times X)$ *of* N *is in the* concurrency relation co *, if neither* $x \leq y$ *nor* $y \leq x$ *nor* $x \# y$*.*
- *For an occurrence net* $N = (P, T, F)$*,* $Min(N)$ *denotes the set of minimal elements of* X *w.r.t.* $\leq$*.*

The relation between a net system $S = (N, M_0)$ with $N = (P, T, F)$ and an occurrence net $O = (C, E, G)$ is defined as a homomorphism $h : C \cup E \mapsto P \cup T$ such that $h(C) \subseteq P$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of h to $\bullet e$ is a bijection between $\bullet e$ and $\bullet h(e)$; the restriction of h to $e\bullet$ is a bijection between $e\bullet$ and $h(e)\bullet$, the restriction of h to $Min(O)$ is a bijection between $Min(O)$ and M_0; and for all $e, f \in E$, if $\bullet e = \bullet f$ and $h(e) = h(f)$ then $e = f$.

A *branching process* of $S = (N, M_0)$ is a tuple $\pi = (O, h)$ with $O = (C, E, G)$ being an occurrence net and h being a homomorphism from O to S as defined above. A branching process $\pi' = (O', h')$ is a *prefix*, if $O' = (C', E', G')$ is a subnet of O, such that if $e \in E'$ and $(c, e) \in G$ or $(e, c) \in G$ then $c \in C'$; if $c \in C'$ and $(e, c) \in G$ then $e \in E'$; h' is the restriction of h to $C' \cup E'$.

The maximal branching process of S is called *unfolding*. The unfolding of a net system can be truncated once all markings of the original net system and all enabled transitions are represented. This yields the complete prefix unfolding.

Definition 4 (Complete Prefix Unfolding). *Let* $S = (N, M_0)$ *be a system and* $\pi = (O, h)$ *a branching process with* $N = (P, T, F)$ *and* $O = (C, E, G)$*.*

- *A set of events* $E' \subseteq E$ *is a* configuration*, iff* $\forall\ e, f \in E'\ [\ \neg(e \# f)\ ]$ *and* $\forall\ e \in E'\ [\ f < e \Rightarrow f \in E'\ ]$*. The* local configuration $\lceil e \rceil$ *for an event* $e \in E$ *is defined as* $\{x \in X \mid x < e\}$*.*
- *A set of conditions* $C' \subseteq C$ *is called* co-set*, iff for all distinct* $c_1, c_2 \in C'$ *it holds* c_1 *co* c_2*. If* C' *is maximal w.r.t. set inclusion, it is called a* cut*.*
- *For a finite configuration* C'*,* $Cut(C') = (Min(O) \cup C' \bullet) \setminus \bullet C'$ *is a cut, while* $h(Cut(C'))$ *is a reachable marking of* S*, denoted* $Mark(C')$*.*
- *The branching process is* complete*, iff for every marking* $M \in [N, M_0\rangle$ *there is a configuration* C' *of* π *such that* $M = Mark(C')$ *and for every transition* t *enabled in* M *there is a finite configuration* C' *and an event* $e \notin C'$ *such that* $M = Mark(C')$*,* $h(e) = t$*, and* $C' \cup \{e\}$ *is a configuration.*
- *An* adequate order $\lhd$ *is a strict well-founded partial order on local configurations such that for two events* $e, f \in E$ $\lceil e \rceil \subset \lceil f \rceil$ *implies* $\lceil e \rceil \lhd \lceil f \rceil$*.*
- *An event* $e \in E$ *is a* cut-off event *induced by* $\lhd$*, iff there is a* corresponding event $f \in E$ *with* $Mark(\lceil e \rceil) = Mark(\lceil f \rceil)$ *and* $\lceil f \rceil \lhd \lceil e \rceil$*.*

- *The branching process π is the* complete prefix unfolding *induced by $\lhd$, iff it is the greatest prefix of the unfolding of S that does not contain any event after a cut-off event.*

We see that the definition of a cut-off event and, therefore, of the complete prefix unfolding is parametrised by the definition of an adequate order $\lhd$. Multiple definitions have been proposed in the literature, cf., [15]. The differences between these definitions can be neglected for our approach and are relevant solely for the experimental evaluation in which we rely on the definition presented in [16]. As we leverage the information on cut-off events in our approach, we include them in the complete prefix for convenience.

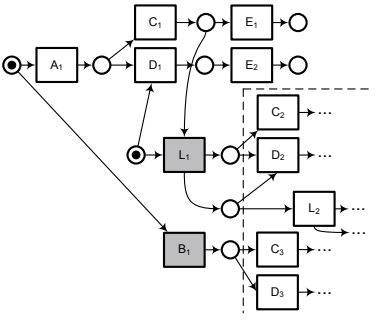


Fig. 2. Complete prefix unfolding of the net system in Fig. 1(b)

Fig. 2 illustrates the concept of an unfolding and its complete prefix for the net system in Fig. 1(b). Here, the labelling of transitions and the initial marking hints at the homomorphism between the two systems. Apparently, the unfolding of the net system is infinite due to the control flow cycle. In Fig. 2, cut-off events are highlighted in grey and the complete prefix unfolding is marked by dashed lines.

2.3 Behavioural Profiles

Behavioural profiles capture behavioural aspects of a system in terms of order constraints [3]. They are based on the set of possible firing sequences of a net system and the notion of *weak order*. Informally, two transitions t_1, t_2 are in weak order, if there exists a firing sequence reachable from the initial marking in which t_1 occurs before t_2.

Definition 5 (Weak Order). *Let (N, M_0) be a net system with $N = (P, T, F)$. Two transitions x, y are in the* weak order relation $\succ \subseteq T \times T$*, iff there exists a firing sequence $\sigma = t_1, \ldots, t_n$ with $(N, M_0)[\sigma\rangle$, $j \in \{1, \ldots, n-1\}$, $j < k \leq n$, for which holds $t_j = x$ and $t_k = y$.*

Depending on how two transitions of a system are related by weak order, we define three relations forming the behavioural profile.

Definition 6 (Behavioural Profile). *Let (N, M_0) be a net system with $N = (P, T, F)$. A pair of transitions $(x, y) \in (T \times T)$ is in at most one of the following* profile relations*:*

- *The* strict order relation $\rightsquigarrow$*, if $x \succ y$ and $y \not\succ x$.*
- *The* exclusiveness relation $+$*, if $x \not\succ y$ and $y \not\succ x$.*
- *The* interleaving order relation $||$*, if $x \succ y$ and $y \succ x$.*

$\mathcal{B} = \{\rightsquigarrow, +, ||\}$ is the behavioural profile *of (N, M_0).*

For our example net system in Fig. 1(b), for instance, it holds $A \rightsquigarrow C$, as there exists no firing sequence, in which C occurs before A. As no firing sequence contains both transitions, A and B, it holds $A + B$. Due to the control flow cycle, it holds $C||L$. That is, both transitions can occur in any order in a firing sequence of the system. With $\rightsquigarrow^{-1}$ as the inverse relation for $\rightsquigarrow$, the relations $\rightsquigarrow$,$\rightsquigarrow^{-1}$,+, and $||$ partition the Cartesian product of transitions.

3 Generalised Computation of Behavioural Profiles

This section introduces our approach of computing the behavioural profile of a bounded net system from its complete prefix unfolding. First, Section 3.1 relates the ordering relations of an unfolding to the relations of the profile. Second, Section 3.2 presents an algorithm for the computation of behavioural profiles. Finally, Section 3.3 presents experimental results.

3.1 Behavioural Relations

The computation of the behavioural profile from the complete prefix unfolding is based on the ordering relations introduced for occurrence nets in Definition 3. The causality, conflict, and concurrency relation partition the Cartesian product of events of an occurrence net and, therefore, of the complete prefix unfolding, similar to the relations of the behavioural profile. However, the ordering relations of an occurrence net relate to *events*, i.e., *occurrences* of transitions of the original net system. Formally, this is manifested in the homomorphism between the net system and its complete prefix unfolding, which may relate multiple events to a single transition. For the net system in Fig. 1(b), for instance, transition C relates to two events in the prefix in Fig. 2, C_1 and C_2 (the latter being a cut-off event). Both events represent an occurrence of transition C in the original system and, hence, have to be considered when deriving the behavioural profile.

In general, the order of potential execution, i.e., the weak order relation of the behavioural profile, can be deduced from the concurrency and the causality relation of the complete prefix unfolding. The existence of a firing sequence containing two transitions of the original system is manifested in two events in the prefix that relate to these transitions and are concurrent or in causality. The former represents two transitions that can be enabled concurrently in the original system. Thus, there is a firing sequence containing both transitions in either order. Two events in causality in the prefix, in turn, represent two transitions in the original net that can occur in a firing sequence in the respective order.

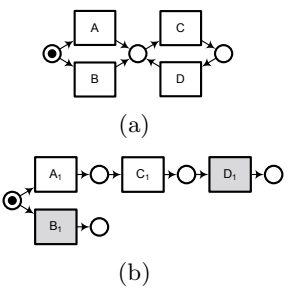


Fig. 3. Net system (a); its complete prefix unfolding (b)

Another observation relates to the fact that not all firing sequences are visible in the complete prefix unfolding directly. Events that relate to two transitions

might not show causality or concurrency although the respective transitions might occur in a firing sequence. Fig. 3 illustrates this issue. Apparently, transition D might be observed in firing sequences that commence with transition B in the net system in Fig. 3(a). The events that represent occurrences of both transitions, B_1 and D_1, are in conflict in the complete prefix unfolding in Fig. 3(b). Hence, the information that is hidden due to the cut of the unfolding has to be taken into account. In Fig. 3(b), firing of B_1 leads to a marking that is already contained in the prefix. That is, firing of A_1 reaches at the same marking and enables the event C_1. Therefore, it can be deduced that an event relating to transition D of the original net can follow any firing of event B_1.

While this example illustrates solely a simple dependency, Fig. 4 illustrates the general case. Consider the events e and f and assume that both events represent the occurrences of different transitions t_e and t_f of the original net system and that both events are in conflict. Still, the transitions t_e and t_f might occur in a firing sequence of the net system due to the cut-off events c_1 and c_2 and their corresponding events c_1' and c_2'. Event e is in a causal relation with the cut-off event c_1. The respective cut of its local configuration is highlighted by a dashed line. This cut corresponds to the cut of the local configuration of event c_1', which, in turn, is concurrent to another cut-off event c_2. Similarly, a relation can be established between c_2 and f via c_2'. This example illustrates that such dependencies between two events may span multiple cut-off events.

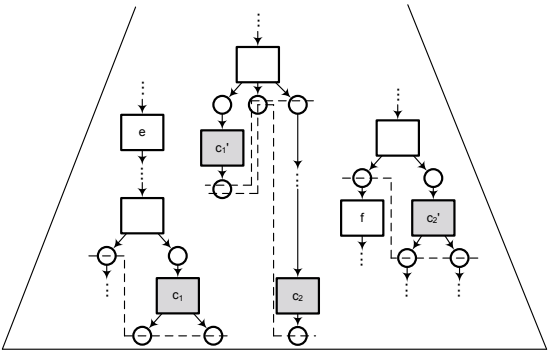


Fig. 4. Sketch of a complete prefix unfolding; e and f can occur together in a firing sequence

Based on this observation, we establish the relation between the ordering relations of the complete prefix unfolding and the weak order relation as follows.

Proposition 1. *Let $S = (N, M_0)$ be a bounded system and $\pi = (O, h)$ its complete prefix unfolding including the cut-off events with $N = (P, T, F)$ and $O = (C, E, G)$. Then, two transitions $x, y \in T$ are in weak order, $x \succ y$, iff there are two events $e, f \in E$ with $h(e) = x$ and $h(f) = y$ and either*

- *they are causally related or concurrent, i.e., $(e < f) \vee (e \; co \; f)$, or*
- *there is a sequence of cut-off events $(g_1, g_2, \ldots, g_n)$ with $g_i \in E$ for $1 \leq i \leq n$ and a sequence of corresponding events $(g_1', g_2', \ldots, g_n')$ with $g_i' \in E$, such that $(e = g_1) \vee (e < g_1)$, $\exists\, c \in Cut(\lceil g_j' \rceil)$ $[\; c < g_{j+1} \;]$ for $1 \leq j < n$, and either $g_n' = f$ or $\exists\, c \in Cut(\lceil g_n' \rceil)$ $[\; c < f \;]$.*

Proof. Let x, y, $(g_1, g_2, \ldots, g_n)$, and $(g_1', g_2', \ldots, g_n')$ be defined as above.
($\Rightarrow$) Let $x \succ y$. Then, there is a reachable firing sequence in S containing

transition x before y. As the prefix is complete, both occurrences of transitions are represented by corresponding events $e, f \in E$ with $h(e) = x$ and $h(f) = y$. If there is a firing sequence in O starting in its initial marking induced by $Min(O)$ that contains e before f, they are causally related or concurrent, i.e., $(e < f) \vee (e \; co \; f)$ (first statement of our proposition). If there is no such firing sequence, then either $e \# f$ or $f < e$. If there are no cut-off events after firing of e in O, all markings that are reached after firing of x in S have their counterparts in π directly. Those do not enable f. Therefore, there has to be a cut-off event k that can occur after firing of e (or which is equal to e). Hence, either $e \leq k$, or $e \; co \; k$. Assume that the sequence of cut-off events is one. Then, the corresponding event k' for the cut-off event k has to occur together with the event f, i.e., $k' < f$, $f < k'$ or $f \; co \; k'$. If $e \; co \; k$, then there has to be an event e' with $h(e) = h(e')$ and $e' \; co \; k'$. In this case, we, again, arrive at $(e' < f) \vee (e' \; co \; f)$. Consider $e \leq k$. To observe a firing of x before y, we have to exclude $f < k'$ from the possible relations between f and k'. That is because $f < k'$ implies that y is observed before the marking represented by the cut of $\lceil k \rceil$ is reached as $Mark(\lceil k \rceil) = Mark(\lceil k' \rceil)$. Hence, we have $k' < f$ or $f \; co \; k'$ (the second statement of our proposition). The same argument can be applied to all intermediate cut-off events in case the sequence of cut-off events is longer than one. ($\Leftarrow$) For all events $e, f \in E$ with $h(e) = x$ and $h(f) = y$ let (1) $(e < f) \vee (e \; co \; f)$, or (2) there is a sequence of cut-off events $(g_1, g_2, \ldots, g_n)$ and corresponding events $(g'_1, g'_2, \ldots, g'_n)$, such that $(e = g_1) \vee (e < g_1)$, $\exists \; c \in Cut(\lceil g'_j \rceil) \; [\; c < g_{j+1} \;]$ for all $j \in \{1, \ldots, n-1\}$, and $g'_n = f$ or $\exists \; c \in Cut(\lceil g'_n \rceil) \; [\; c < f \;]$. Assume that $x \not> y$. From the initial marking of the occurrence net that is induced by $Min(O)$, we can fire all events of the local configuration of f, $\lceil f \rceil$, in their order induced by $<$ to reach a marking M_1. Then, event f can be fired in this marking to reach marking M_2. If $e < f$ then event is e is part of $\lceil f \rceil$ and, therefore, has been fired already. If $e \; co \; f$, event e has not been fired as part of $\lceil f \rceil$ to reach M_1. Still, there must be an event g in $\lceil f \rceil$ such that for every condition $c_f \in \bullet f$ there is a condition $c_g \in g\bullet$ with $c_g < c_f$ or $c_g = c_f$. Due to $e \; co \; f$, $g < f$, and $g < e$, we also know $c_g \not< e$ for all those conditions c_g. All conditions c_g are marked in M_1. They are also marked in M_2 reached via firing of e as $c_g \not< e$. Hence, there has to be a firing sequence starting in M_2 and containing all events that are part of $\lceil f \rceil \setminus \lceil e \rceil$. Then, event f can be fired in the reached marking. Thus, in both cases there is a firing sequence containing both events e and f. When all fired events are resolved according to h, there is a firing sequence in S containing x before y, which is a contradiction with $x \not> y$. Consider case (2). Following on the argument given in the previous case, we know that there is a firing sequence in the occurrence net that contains event e before event g_1. For the corresponding event g'_1, we know $Mark(\lceil g_1 \rceil) = Mark(\lceil g'_1 \rceil)$. Hence, the marking in S reached via firing the transitions that correspond to all events $\lceil g_1 \rceil$ is equal to the marking reached by firing the corresponding transitions in $\lceil g'_1 \rceil$. Assume that the sequence of cut-off events is one, i.e., $\exists \; c \in Cut(\lceil g'_1 \rceil) \; [\; c < f \;]$. Again, there is a firing sequence in the occurrence net that contains g'_1 before event f. Due to $Mark(\lceil g_1 \rceil) = Mark(\lceil g'_1 \rceil)$, there is a firing sequence in S that

contains the transition represented by the event g_1 before the one represented by f, i.e., transition y. From $e < g_1$, we get that such a firing sequence can contain the transition represented by event e, i.e., x, before the one represented by g_1 and, therefore, also before y. Thus, there is a firing sequence in S in which x is followed by y and we arrive at a contradiction with $x \not\prec y$. Again, the same argument can be applied to all intermediate cut-off events in case the sequence of cut-off events is longer than one. □

3.2 Computation Algorithm

As we have seen in the previous section, the weak order relation of the behavioural profile can be traced back to the ordering relations of the complete prefix unfolding. Based thereon, Algorithm 1 shows how the behavioural profile is computed for a bounded system given its complete prefix unfolding.

First and foremost, the algorithm comprises the computation of the ordering relations, i.e., the causality, conflict, and concurrency relation, for the complete prefix unfolding (line 1). The respective algorithm can be found in [17].

Second, we capture relations between cut-off events (lines 2 to 12). We compute the relation between events and the conditions that belong to the cut induced by their local configuration (set $\mathcal{L}$). Then, causality between these conditions of one event and another event is captured in set $\mathcal{LC}$. The set $\mathcal{C}_{cut}$ is filled with all cut-off events, while their corresponding events are added to the set $\mathcal{C}_{cor}$. The relation between them is stored as an entry in the relation $\mathcal{C}$. We check for each event that corresponds to a cut-off event, whether it is in a causal relation with a condition of the cut relating to the local configuration of another cut-off event. If so, this information is stored in the relation $\mathcal{C}$. The intuition behind is that the transitive closure of $\mathcal{C}$ hints at the existence of a sequence of cut-off and corresponding events as defined in Proposition 1.

Third, all pairs of events of the complete prefix unfolding are assessed according to Proposition 1 (lines 13 to 21). If the respective requirements are met, the weak order relation is captured for the transitions that are represented by these events.

Finally, the relations of the behavioural profile are derived from the weak order relation according to Definition 6 (lines 22 to 27).

Proposition 2. *Algorithm 1 terminates and after termination* $\mathcal{B} = \{\rightsquigarrow, +, ||\}$ *is the behavioural profile of* S.

Proof. Termination: The algorithm iterates over sets that are derived from E, C, $\mathcal{C}_{cut} \subseteq E$, $\mathcal{C}_{cor} \subseteq E$, and T. T is finite by definition. Due to boundedness of the net system the complete prefix unfolding and, therefore, the sets of events E and conditions C are finite as well. Hence, the algorithm terminates.

Result: The set $\mathcal{L}$ contains all pairs of events and conditions, such that the condition belongs to the cut of the local configuration of the respective event. That is achieved by considering all conditions of the postset of the event and all concurrent conditions that are either initially marked or are part of a postset of another event that is in a causal relation with the former event. Then, an entry of set $\mathcal{LC}$ associates an event to all events that are in causality to one of

Algorithm 1. Algorithm for the computation of the behavioural profile

Input: $S = (N, M_0)$, a bounded system with $N = (P, T, F)$.
$\pi = (O, h)$, its complete prefix unfolding including cut-off events with $O = (C, E, G)$.
Output: $\mathcal{B} = \{\rightsquigarrow, +, ||\}$, the behavioural profile of S.

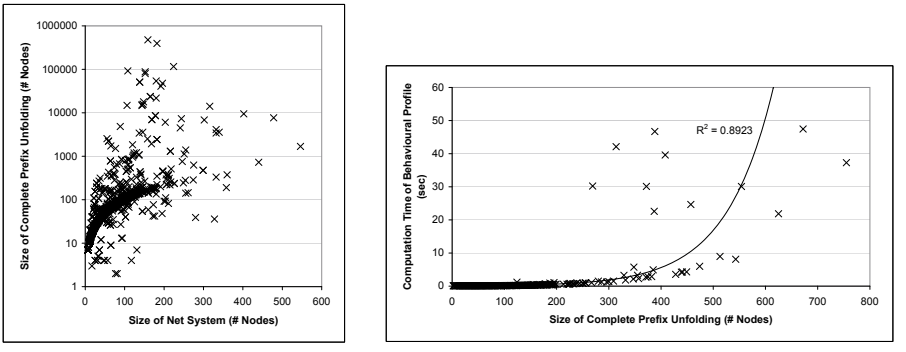
Compute ordering relations $<$, $\#$, and co of O;
/* Establish relation between cut-off events of O */
$\mathcal{L}, \mathcal{LC}, \mathcal{C}_{cor}, \mathcal{C}_{cut}, \mathcal{C} \longleftarrow \emptyset$;
foreach $(e, c) \in (E \times C)$ **do**
  if $(c \in e\bullet) \vee ((e\ co\ c) \wedge ((\bullet c \times \{e\} \subseteq <) \vee (\bullet c = \emptyset)))$ **then** $\mathcal{L} \longleftarrow (e, c)$;
end
foreach $(e_1, c, e_2) \in (\mathcal{L} \times E)$ **do** **if** $c < e_2$ **then** $\mathcal{LC} \longleftarrow (e_1, e_2)$;
foreach $(e_1, e_2) \in (E \times E)$ **do**
  if $(Mark(\lceil e_1 \rceil) = Mark(\lceil e_2 \rceil)) \wedge (\lceil e_1 \rceil \lhd \lceil e_2 \rceil)$ **then**
    $\mathcal{C}_{cor} \longleftarrow (e_1)$; $\mathcal{C}_{cut} \longleftarrow (e_2)$; $\mathcal{C} \longleftarrow (e_2, e_1)$;
  end
end
foreach $(e_{cor}, e_{cut}) \in (\mathcal{C}_{cor} \times \mathcal{C}_{cut})$ **do** **if** $e_{cor}\ \mathcal{LC}\ e_{cut}$ **then** $\mathcal{C} \longleftarrow (e_{cor}, e_{cut})$;
/* Derive weak order for transitions of N */
$\succ \longleftarrow \emptyset$;
foreach $(e_1, e_2) \in (E \times E)$ **do**
  if $(e_1 < e_2) \vee (e_1\ co\ e_2)$ **then** $\succ \longleftarrow (h(e_1), h(e_2))$;
  else foreach $(e_{cut}, e_{cor}) \in (\mathcal{C}_{cut} \times \mathcal{C}_{cor})$ **do**
    if $(e_1 \leq e_{cut}) \wedge (e_{cut}\ \mathcal{C}^+\ e_{cor}) \wedge (e_{cor}\ \mathcal{LC}\ e_2))$ **then**
      $\succ \longleftarrow (h(e_1), h(e_2))$;
    end
  end
end
/* Derive relations of behavioural profile of N */
$\rightsquigarrow, +, ||\ \longleftarrow \emptyset$;
foreach $(t_1, t_2) \in (T \times T)$ **do**
  if $(t_1 \succ t_2) \wedge (t_2 \succ t_1)$ **then** $||\ \longleftarrow (t_1, t_2)$;
  else if $t_1 \succ t_2$ **then** $\rightsquigarrow \longleftarrow (t_1, t_2)$;
  else $+\ \longleftarrow (t_1, t_2)$;
end

the conditions of the former event's cut. Moreover, set $\mathcal{C}$ is build such that it contains all pairs of cut-off events and their corresponding events. Then, pairs of corresponding events and further cut-off events are added, if there is a causal relation between them in $\mathcal{LC}$. Hence, the transitive closure of $\mathcal{C}$ hints at the existence of a sequence of cut-off events and corresponding events that are causally related. Based thereon, Proposition 1 is implemented directly. The derivation of the profile from the weak order relation follows directly on Definition 6. □

The algorithm runs in polynomial time with respect to size of the complete prefix unfolding. The final step of the algorithm that sets the profile relations based on the weak order relation for all pairs of transitions is neglected at this point.

Proposition 3. *The following problem can be solved in $O(n^4)$ time with n as the number of events and conditions of the complete prefix unfolding:*
For a bounded net system and its complete prefix unfolding, to compute the weak order relation for the net system.

Proof. Computation of the ordering relations of the complete prefix unfolding can be done in $O(|E| * |C|)$ time [17]. In the second step of the algorithm, we iterate over $E \times C$, $E \times C \times E$, and $E \times E$, which takes $O(|E|^2 * |C|)$ time. Due to $\mathcal{C}_{cut} \subseteq E$ and $\mathcal{C}_{cor} \subseteq E$, the third step takes $O(|E|^4)$ time. As a prerequisite



(a) (b)

Fig. 5. (a) Size of complete prefix unfolding relative to the size of the net system; (b) overall computation time relative to the size of the complete prefix unfolding

for this step the transitive closure of $\mathcal{C}$ has to be computed, which takes $O(|E|^3)$ time. Therefore, overall time complexity is $O(n^4)$ with n as the number of events and conditions of the complete prefix unfolding. □

3.3 Experimental Evaluation

The approach introduced in the previous section is applicable in a general case as it requires only boundedness of the net system. However, the generality is traded for computational complexity. On the one hand, computation of the prefix unfolding is computationally hard, as it is an NP-complete problem. On the other hand, our algorithm runs in polynomial time with respect to the size of the complete prefix unfolding. We are using the partial order defined in [16] for identifying cut-off events, which has been shown to create compact prefixes. Nevertheless, complete prefix unfoldings might have a very large size, such that even a polynomial time algorithm results in long computation times. According to [16], the size of the prefix is at most the size of the reachability graph.

In order to investigate the implications of these issues, we conducted an experiment based on 735 industry process models for which net systems have been derived (see [7] for further details). Although these systems were free-choice, only half of them were sound [7]. Hence, the existing efficient algorithms would be applicable solely for half of these models. In the implementation of our approach, we used Mole[1] to generate the complete prefix unfolding.

For four net systems, creation of the complete prefix unfolding was not possible due to the size of the unfolding. The maximum size of the derived complete prefix unfoldings was 500.000 nodes. Fig. 5(a) illustrates that prefixes can be an order of magnitude larger in size than the original net systems (note the logarithmic scale). Still, the majority of complete prefix unfoldings was rather small. 94% of the net systems had a prefix with less than 800 nodes. For these net systems, Fig. 5(b) shows the overall computation time of the behavioural profile including prefix creation relative to the size of the prefix in terms of the number of nodes. For

[1] http://www.fmi.uni-stuttgart.de/szs/tools/mole/

small prefixes with less than 250 nodes, computation of the behavioural profile is done within one or two seconds. However, computation for prefixes with more than 300 nodes may take up to tens of seconds. The inherent exponential complexity is also visible in the exponential least squares regression depicted in Fig. 5(b). Taking into account that 94% of the net systems have a complete prefix unfolding with less than 800 nodes, we see that our approach works for the majority of process models in the collection within a reasonable time.

In order to shade light on the trade-off between efficiency and generality of the different approaches for the profile computation, Fig. 6 compares the generic approach based on unfoldings with the structural approach presented in [3] (along with exponential or polynomial least squares regressions). Due to the assumptions of the latter approach, solely sound systems are considered. The differences in the base effort are due to the usage of an external unfolding tool. We see that effects in terms of efficiency become visible for net systems that have more than 100 nodes. For these systems, the structural characterisation building on the assumption of soundness and free-choiceness is more efficient.

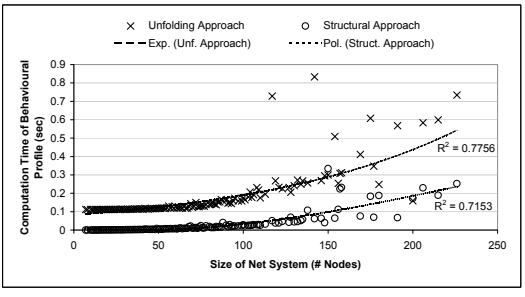


Fig. 6. Overall computation time relative to the size of the net system

4 Behavioural Profile of a Labelled Net System

As the final step of our generalised profile computation, we lift the concept to the level of labelled net systems. While our initial example in Fig. 1 comprises solely unique transitions, this assumption does not hold for real world process models. Multiple transitions with the same label occur as activities might be executed at different stages of a process. For instance, exception handling for an activity might comprise the possibility to redo the activity. Then, there may be two transitions with the same label, one as part of the standard processing and one as part of the exception handling.

Formally, a *labelled net* is a tuple $N = (P, T, F, \Lambda, \lambda)$, where (P, T, F) is a net, Λ is a set of labels, and $\lambda : T \mapsto \Lambda$ is a labelling function. (N, M_0) is called *labelled* system, iff N is a labelled net. The weak order relation can be lifted to labels as follows. Two labels are in weak order, if and only if, two transitions carrying those labels are in weak order. Based thereon, the relations of the behavioural profile can be lifted to labels in a straight-forward manner.

Definition 7 (Weak Order & Behavioural Profile on Labels). *Let $S = (N, M_0)$ be a labelled system with $N = (P, T, F, \Lambda, \lambda)$ and $\succ$ the weak order relation of S.*

- *Two labels l_1, l_2 are in the* weak order on labels relation $\succ_\Lambda \subseteq \Lambda \times \Lambda$*, iff there are two transitions $x, y \in T$ such that $\lambda(x) = l_1$, $\lambda(y) = l_2$, and $x \succ y$.*

- *A pair of labels* $(l_1, l_2) \in (\Lambda \times \Lambda)$ *is in at most one of the following relations:*
  - *The* strict order relation $\rightsquigarrow_\Lambda$*, if* $l_1 \succ_\Lambda l_2$ *and* $l_2 \not\succ_\Lambda l_1$*.*
  - *The* exclusiveness relation $+_\Lambda$*, if* $l_1 \not\succ_\Lambda l_2$ *and* $l_2 \not\succ_\Lambda l_1$*.*
  - *The* interleaving order relation $||_\Lambda$*, if* $l_1 \succ_\Lambda l_2$ *and* $l_2 \succ_\Lambda l_1$*.*

  $\mathcal{B} = \{\rightsquigarrow_\Lambda, +_\Lambda, ||_\Lambda\}$ *is the* behavioural profile on labels *of* S*.*

As behavioural profile on labels is derived directly from the behavioural profile its computation can be done efficiently.

Proposition 4. *The following problem can be solved in* $O(n^2)$ *time with* n *as the number of transitions:*
Given the behavioural profile of a labelled net system, to derive its behavioural profile on labels.

Proof. Both, deriving the weak order relation on labels and setting the relations of the behavioural profile on labels, requires iteration over the Cartesian product of transitions of the net system. Assuming that each label relates to at least one transition, the number of labels is smaller than the number of transitions. Thus, overall time complexity is $O(n^2)$ with n as the number of transitions. □

5 Related Work

Since the unfolding technique has been introduced by McMillan [14], it has been extended and analysed in a huge number of publications, see [15] for a thorough discussion. Moreover, this technique has been applied for various purposes. Unfoldings are used to check common properties of net systems, such as reachability of certain markings, or even for LTL model checking (cf., [18]). Moreover, domain specific problems, e.g., the analysis and synthesis of asynchronous circuits [19], have been addressed using the unfolding technique. Recently, complete prefix unfoldings have been used to restructure process models [20].

Besides applications of unfoldings, our work relates to notions of behavioural consistency or similarity, as behavioural profiles provide a behavioural abstraction. There is a huge body of work on such notions, starting with the equivalence criteria of the linear time – branching time spectrum [21]. Similarity measures may define an edit distance between processes, which, in turn, is based on the n-gram representation of the process language or the underlying automaton [22]. Close to behavioural profiles are causal footprints [23]. These footprints capture causal dependencies between activities and can also be leveraged to determine behavioural similarity. Behavioural relations between activities are also at the core of many process mining algorithms that aim at constructing models from event logs. The α-algorithm for mining process models [24] is based on relations that are similar to those of the behavioural profile, yet different.

6 Conclusion

In this paper, we generalised the computation of behavioural profiles. We introduced an algorithm that derives behavioural profiles from the complete prefix unfolding of a bounded Petri net. Moreover, we showed how behavioural

profiles are lifted to the level of labelled net systems. Therefore, our approach is applicable in a more general setting than the existing techniques.

The overall complexity of our approach is dominated by the exponential complexity of computing the prefix unfolding, which is an NP-complete problem. We investigated the implications of this complexity issue by an experimental setup involving 735 industry process models. Our results show that computation of behavioural profiles is feasible within seconds for a certain class of process models, for which the complete prefix unfolding contains several hundreds of nodes. We also proved that our computation based on the complete prefix unfolding runs in low polynomial time. Due to the inherent complexity of the unfolding technique, our approach cannot be applied to all process models. However, we significantly broadened the set of process models for which behavioural profiles can be derived as existing algorithms assume soundness and free-choiceness of net systems. Therefore, our approach gives rise to the usage of the various applications of behavioural profiles (cf., [3,5,6,25]) for a wider class of process models.

There is a lot of potential for the combination of the presented approach with structural decomposition techniques, similar to those used in [4] for the computation of behavioural profiles for sound free-choice systems. Sub-systems that represent single-entry single-exit blocks might be considered in isolation. Then, the profile is computed iteratively for all these sub-systems, which in combination yield the profile for the whole system. The unfoldings of these sub-systems may be significantly smaller in size than the unfolding of the overall system. For sound and free-choice sub-systems, the efficient algorithms from our previous work can be used. We want to investigate the combination of these techniques in future work. Moreover, we want to explore the application of unfoldings for unbounded net systems [26], which would allow for the computation of behavioural profiles even for this class of systems.

Acknowledgements. We would like to thank the anonymous reviewers for the constructive comments. We very much appreciate their suggestions regarding the experimental evaluation and the simplification of the proof of Proposition 1.

References

1. Dumas, M., García-Bañuelos, L., Dijkman, R.M.: Similarity search of business process models. IEEE Data Eng. Bull. 32(3), 23–28 (2009)
2. van der Aalst, W.M.P.: Inheritance of business processes: A journey visiting four notorious problems. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 383–408. Springer, Heidelberg (2003)
3. Weidlich, M., Mendling, J., Weske, M.: Efficient consistency measurement based on behavioural profiles of process models. IEEE Transactions on Software Engineering (2010) (to appear)
4. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Efficient computation of causal behavioural profiles using structural decomposition. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 63–83. Springer, Heidelberg (2010)
5. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J.: Process compliance measurement based on behavioural profiles. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 483–498. Springer, Heidelberg (2010)

6. Weidlich, M., Weske, M., Mendling, J.: Change propagation in process models using behavioural profiles. In: IEEE SCC, pp. 33–40. IEEE CS, Los Alamitos (2009)
7. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 278–293. Springer, Heidelberg (2009)
8. Martens, A.: On Compatibility of Web Services. Petri Net Newsletter 65, 12–20 (2003)
9. Dijkman, R., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. IST 50(12), 1281–1294 (2009)
10. Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
11. Kindler, E.: On the semantics of EPCs: A framework for resolving the vicious circle. In: Desel, J., Pernici, B., Weske, M. (eds.) BPM 2004. LNCS, vol. 3080, pp. 82–97. Springer, Heidelberg (2004)
12. Lohmann, N., Verbeek, E., Dijkman, R.M.: Petri net transformations for business processes - a survey. TOPNOC 2, 46–63 (2009)
13. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) Petri Nets1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
14. McMillan, K.L.: A technique of state space search based on unfolding. FMSD 6(1), 45–65 (1995)
15. Esparza, J., Heljanko, K.: Unfoldings: a partial-order approach to model checking. Springer, Heidelberg (2008)
16. Esparza, J., Römer, S., Vogler, W.: An improvement of mcmillan's unfolding algorithm. FMSD 20(3), 285–310 (2002)
17. Kondratyev, A., Kishinevsky, M., Taubin, A., Ten, S.: Analysis of petri nets by ordering relations in reduced unfoldings. FMSD 12(1), 5–38 (1998)
18. Esparza, J., Heljanko, K.: A new unfolding approach to ltl model checking. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 475–486. Springer, Heidelberg (2000)
19. Khomenko, V., Koutny, M., Yakovlev, A.: Logic synthesis for asynchronous circuits based on stg unfoldings and incremental sat. Fundam. Inform. 70(1-2), 49–73 (2006)
20. Polyvyanyy, A., García-Bañuelos, L.L., Dumas, M.: Structuring acyclic process models. In: BPM (September 2010) (to appear)
21. Glabbeek, R., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Inf. 37(4/5), 229–327 (2001)
22. Wombacher, A.: Evaluation of technical measures for workflow similarity based on a pilot study. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 255–272. Springer, Heidelberg (2006)
23. Dongen, B., Dijkman, R.M., Mendling, J.: Measuring similarity between business process models. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 450–464. Springer, Heidelberg (2008)
24. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE TKDE 16(9), 1128–1142 (2004)
25. Smirnov, S., Weidlich, M., Mendling, J., Weske, M.: Action patterns in business process models. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 115–129. Springer, Heidelberg (2009)
26. Desel, J., Juhás, G., Neumair, C.: Finite unfoldings of unbounded petri nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 157–176. Springer, Heidelberg (2004)

Constructing Replaceable Services Using Operating Guidelines and Maximal Controllers

Arjan J. Mooij[1], Jarungjit Parnjai[2], Christian Stahl[1], and Marc Voorhoeve[1]

[1] Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
{A.J.Mooij,C.Stahl,M.Voorhoeve}@tue.nl
[2] Humboldt-Universität zu Berlin, Institut für Informatik, Berlin, Germany
parnjai@informatik.hu-berlin.de

Abstract. Service-oriented systems support process evolution by allowing the replacement of a service S by another service T. To maintain proper interaction in the overall system, service T should interact properly with all controllers (i. e., in all contexts) of service S.

To support the construction of such services T, we compute operating guidelines that *represent all services that can replace service* S. Our computation relies on the additional notion of a *maximal controller*. Maximal controllers can also be used for deciding whether a service T can replace service S, and for computing a public view that hides service details that are not relevant for controllers.

1 Introduction

Complex business processes typically combine several simpler processes that collaborate across the boundaries of organizations. Such collaborations inevitably evolve over time, e. g., because one organization wants to implement a new functionality, or another one wants to improve an existing functionality.

Service-orientation supports process evolution by considering a complex business process as a collaboration of several simpler, interacting services. Replacing one or more of these services, however, may endanger the proper interaction in unexpected ways. As usually no single organization can oversee the full collaboration, verifying the collaboration is not a feasible option.

In this paper, we address the replacement of services that are stateful rather than stateless, i. e., their exposed operations have to be invoked in a particular order, described by their business protocol. To this end, we focus on *protocol changes* [24], and ignore issues like nonfunctional properties.

A service T can *replace* a service S if every admissible context for service S is also admissible for T. Admissible contexts are defined as *controllers*; service R is called a controller of service S if R and S can interact in a deadlock-free manner. This notion of replacement is formalized by the *accordance* preorder [27]: we say that T *accords with* S if every controller of S is a controller of T.

Constructing nontrivial replaceable services is an activity that is vital for many organizations to stay competitive, but also known to be error-prone and

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 116–130, 2011.

time-consuming. Yet, the BPM tools that are currently available on the market offer only limited support. The language WS-BPEL, for example, has rules (called profiles) allowing to transform a service S into a service T that can replace S, but these rules are very restricted. Even the extensions proposed in [14] are incomplete, as they do not cover all possible replacements.

Our main contribution is to support the construction of a service T that can replace S, by computing the set of all services that can replace a given service S. We represent such an infinite set using operating guidelines [19]. Our solution relies on the additional, but related, notion of a *maximal controller*. A maximal controller $mc(S)$ of a service S is a controller of S with the property that every controller of S accords with $mc(S)$. This notion differs from a most-permissive controller [30] of S, which is a controller that exhibits the behavior of every controller of S. A maximal controller of S can be seen as a single service that encodes the set of all controllers of S. Moreover, every controller T of a maximal controller of S can replace service S.

There are at least two other applications of maximal controllers. One is deciding whether a given service T can replace a given service S. The other is computing a public view of a service S that hides all service internals that are not relevant for controllers of S.

Overview. Section 2 continues with some background. In Sect. 3 we study accordance. We explore maximal controllers in Sect. 4 and show in Sect. 5 how they can be constructed. In Sect. 6 we present applications of maximal controllers. Finally Sect. 7 discusses related work, and Sect. 8 concludes the paper.

2 Preliminaries

In this section, we describe our service model, service automata, the notion of a controller, accordance, and operating guidelines.

2.1 Services and Controllers

A service consists of a control structure describing its *behavior* and an interface for asynchronous communication with other services. An interface is a set of (input and output) *channels*. We abstract from the syntax of service description languages and use service automata to model service behavior.

An *automaton* (Q, q_0, I, O, δ) consists of a finite set Q of states, an initial state $q_0 \in Q$, a set I of input channels, a set O of output channels (I and O are disjoint and do not contain τ), and a transition relation $\delta \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$. An automaton is *deterministic* if τ only occurs in selfloops and no state has any two equally labeled outgoing transitions. A *service automaton* (or *service*, for short) $(Q, q_0, \Omega, I, O, \delta)$ consists of an automaton (Q, q_0, I, O, δ) and a set $\Omega \subseteq Q$ of final states [19].

An m-labeled transition $(q, m, q') \in \delta$ is a sending transition if $m \in O$, a receiving transition if $m \in I$, and an internal transition if $m = \tau$. In the graphical representation, we label sending a message to a channel m as $!m$ and receiving

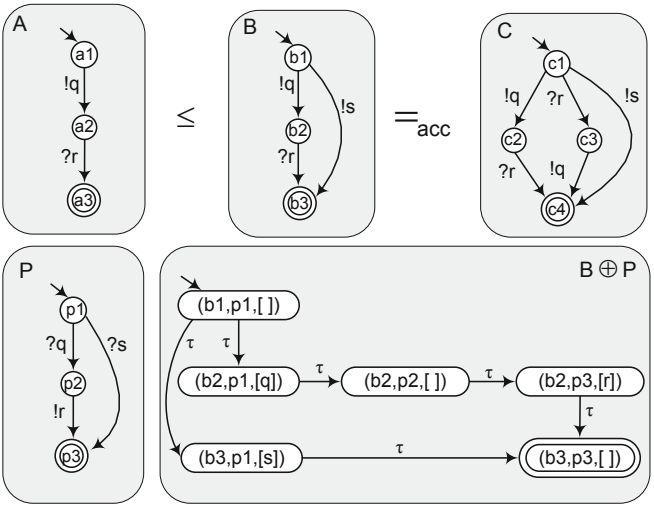


Fig. 1. Running example

a message from a channel m as $?m$. An arc without source state indicates the initial state, and double circles indicate the final states.

We can *compose* two services R and S if the input channels of R are the output channels of S, and vice versa. Composition yields a service $R \oplus S$ where states are triples $(q_R, q_S, \mathcal{M})$ consisting of a state q_R of R, a state q_S of S, and a multiset (i. e., bag) $\mathcal{M}$ of currently pending asynchronous messages that were sent but not yet received. The initial state $(q_{0R}, q_{0S}, [\,])$ consists of the two initial states $q_{0\,R}$ and $q_{0\,S}$ of R and S, and the empty multiset $[\,]$. An m-labeled sending transition of R adds[1] in $R \oplus S$ one element m to the bag $\mathcal{M}$. Similarly a transition receiving an m removes one m from $\mathcal{M}$. Internal transitions of R and S do not update $\mathcal{M}$. In the composition all transitions become internal transitions labeled τ. Final states of the composed system are those where both services are in their respective final states, and the message bag is empty.

Two services R and S interact properly if their composition $R \oplus S$ is *deadlock-free* (denoted by $DF(R \oplus S)$); that is, every reachable nonfinal state has an outgoing transition. In this case, R is a *controller* of S. The set $\mathit{Controllers}(S) = \{R \mid DF(R \oplus S)\}$ denotes the set of all controllers of S. If S has at least one controller, then S is *controllable*. The notion of a controller is symmetric; so if R is a controller of S, then S is a controller of R.

Example 1. Figure 1 depicts the business protocols of customers A, B, and C, each represented as a service with the same interface. All customers can send a request (labeled $!q$) and then receive a response (labeled $?r$); customer C can additionally receive a response before sending a request. Alternatively, customers

[1] We ensure that $R \oplus S$ has again finitely many states by disallowing m-labeled sending transitions if there are k messages m in M; see [19]. In our examples we use $k = 1$, but our theoretical results are independent of any specific k.

B and C can send a stop message (labeled $!s$) at the beginning. The composition $B \oplus P$ of customer B and online shop P (as depicted) is deadlock-free. Hence B is a controller of P, and vice versa, and B and P are controllable.

For comparing two automata S and T, we use a *simulation relation* [20]. Automaton $T = (Q_T, q_{0T}, I, O, \delta_T)$ simulates automaton $S = (Q_S, q_{0S}, I, O, \delta_S)$ if there exists a binary relation $\varrho \subseteq Q_S \times Q_T$ where $(q_{0S}, q_{0T}) \in \varrho$ and, for all $(q_S, q_T) \in \varrho$ and $(q_S, m, q'_S) \in \delta_S$, there exists a state q'_T such that $(q_T, m, q'_T) \in \delta_T$ and $(q'_S, q'_T) \in \varrho$. If T simulates S using a relation ϱ, and S simulates T using relation ϱ^{-1}, then S and T are bisimilar [25], denoted by $S =_{bsim} T$. These notions can be lifted to services, by requiring the simulation relation ϱ to be such that also for all $(q_S, q_T) \in \varrho$ if q_S is a final state then q_T is a final state.

2.2 Accordance Preorder and Equivalence

A service T can replace a service S, if every controller of S is a controller of T. This replaceability notion is *accordance* [27], and technically it is a preorder on services. Using the notations from [21], we formalize accordance $\leq$ as for all services S and T: $T \leq S \;\Leftrightarrow\; \mathit{Controllers}(S) \subseteq \mathit{Controllers}(T)$. The set $\mathit{Accord}(S) = \{T \mid T \leq S\}$ contains all services T that accord with S. The accordance preorder $\leq$ induces an equivalence relation $=_{acc}$ that relates services with identical sets of controllers; for all services S and T: $S =_{acc} T \;\Leftrightarrow\; S \leq T \wedge T \leq S$.

Example 2. In Fig. 1, $A \leq B$ and $B =_{acc} C$. To show that $B \leq A$ does not hold, define P' as P without the transition $?s$. Service P' is a controller of A but not of B, as the state $(b3, p'1, [s])$ is a deadlock in the composition $B \oplus P'$.

2.3 Operating Guidelines

If a service S is controllable, then it has a *most-permissive controller* [30]. There may exist more than one most-permissive controller, but each of them can exhibit all behavior that any controller of S can exhibit. This gives a necessary condition for deciding whether a service R is a controller of S: a most-permissive controller of S must simulate R.

To obtain an exact characterization of the set $\mathit{Controllers}(S)$, we annotate each state q of the most-permissive controller with a Boolean formula $\phi(q)$. These formulae consists of conjunctions $\wedge$, disjunctions $\vee$, and atomic propositions $I_S \cup O_S \cup \{\tau, \mathit{final}\}$, indicating for a state whether certain outgoing edges are present and whether the state is final. The *operating guidelines* of S [19] is the annotated automaton $OG(S) = (Z_S, \phi)$, where Z_S denotes the automaton of the most-permissive controller of S. Note that Z_S has no final states; this information is encoded in the formulae instead.

To determine whether a service R is a controller of S, we analyze whether R matches with $OG(S)$, denoted by $R \in \mathit{Match}(OG(S))$. Service R *matches* with $OG(S)$ if R has the same interface as Z_S and there is a simulation relation

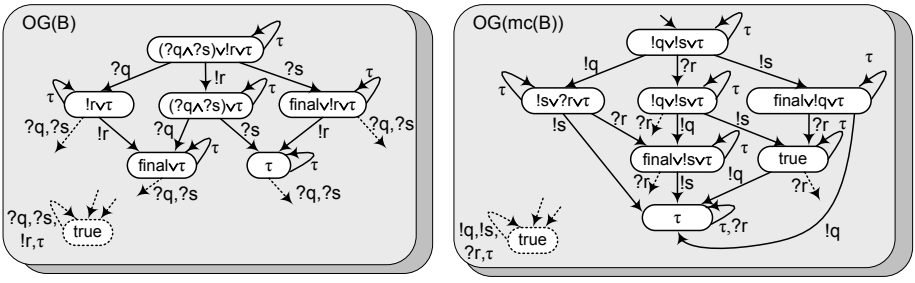


Fig. 2. Operating guidelines for the running example

$\varrho \subseteq Q_R \times Q_Z$ such that, for all $(q_R, q_Z) \in \varrho$ the formula $\phi(q_Z)$ is satisfied in the following assignment β (denoted by $\beta \models \phi(q_Z)$). Assignment β is a Boolean function on $I_S \cup O_S \cup \{\tau, \mathit{final}\}$ such that $\beta(x)$, for $x \in I_S \cup O_S \cup \{\tau\}$, is true if there exists a state q'_R with $(q_R, x, q'_R) \in \delta_R$, and $\beta(\mathit{final})$ is true if $q_R \in \Omega_R$.

Proposition 1 ([19]). *For each service S, $\mathit{Match}(OG(S)) = \mathit{Controllers}(S)$.*

From [19], we conclude that most-permissive controllers, and hence also operating guidelines, have a deterministic transition relation. Moreover, if the simulation relates states q_R and q_Z, and q_R enables a τ-transition in R, then the formula $\phi(q_Z)$ is satisfied by state q_R.

We can use operating guidelines to decide whether T accords with S, i. e., $T \leq S$, by relating the operating guidelines $OG(S) = (Z_S, \phi)$ and $OG(T) = (Z_T, \psi)$: Z_T must simulate Z_S (i. e., Z_T simulates every controller of S) such that the formulae annotated to every state in $OG(S)$ imply the annotated formulae of the respective state in $OG(T)$ (i. e., whenever a service R deadlocks with T it does so with S).

Proposition 2 ([27]). *For each two services S and T, with $OG(S) = (Z_S, \phi)$ and $OG(T) = (Z_T, \psi)$, we have that $\mathit{Controllers}(S) \subseteq \mathit{Controllers}(T)$ iff there is a simulation relation $\varrho \subseteq Q_{Z_S} \times Q_{Z_T}$ such that, for all $(q, q') \in \varrho$, the formula $(\phi(q) \implies \psi(q'))$ is a tautology.*

Example 3. Figure 2 depicts operating guidelines $OG(B)$ of customer B in Fig. 1. Every dashed edge leaving a state has the dashed node as its target. A most-permissive controller of B can receive the corresponding messages but they will never occur, because B cannot send them. The annotation of the initial state shows that a controller must be able to receive both a response *and* a stop message, *or* send a request, *or* do an internal action. In the τ-annotated state, B is in its final state $b3$ and one response message is pending. Hence, a controller has to continuously execute τ-steps to avoid a deadlock in the composition (as indicated by the annotation). Online shop P in Fig. 1 matches with $OG(B)$: the automaton underlying $OG(B)$ simulates P, and the states of P satisfy the annotations. For example, state $p1$ corresponds to an assignment β that only assigns *true* to *?q* and *?s*, and this satisfies the annotation in the initial state of the operating guidelines.

3 Accordance-Preserving Transformation Rules

Our final goal is to compute operating guidelines representing $Accord(S)$ for a service S. Before we can discuss this, we first need two rules that transform a service S into a service T that can replace it. These rules can be seen as an extension of the transformation rules from [1,14] and have the style of the Murata rules [23]. These rules, like the ones from [1], make the services smaller or bigger with respect to accordance.

Rule 1 from Fig. 3(a) specifies that an intermediate τ-transition can be inserted into a service S or removed from a service T, provided that:

- states s_1 and t_1 have the same incoming transitions,
- states s_1 and t_2 have the same outgoing transitions, and
- s_1 is a final state if and only if t_2 is a final state.

Proposition 3 (Rule 1 [23,22]). *For each two services S and T that are related as in Fig. 3(a), it holds that $S =_{acc} T$.*

Rule 2 from Fig. 3(b) specifies that a service becomes smaller with respect to $\leq$ by removing alternatives for an outgoing τ-transition, provided that:

- states t_1 and s_1 have the same incoming transitions,
- the outgoing transitions of state t_1 are contained in those of state s_1,
- states t_2 and s_2 have the same outgoing transitions, and
- t_2 is a final state if and only if s_2 is a final state.

Theorem 1 (Rule 2). *For each two services T and S that are related as in Fig. 3(b), it holds that $T \leq S$.*

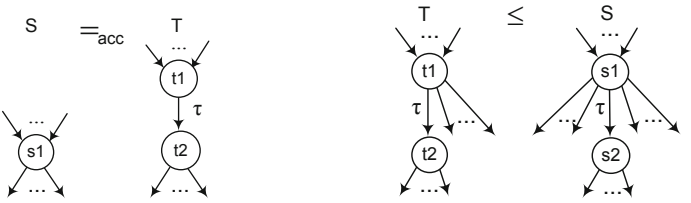
Proof. Suppose the composition $X \oplus T$ (for any service X) can reach a state that is not final and that has no outgoing transition. In this state, T is not in t_1, as t_1 has an outgoing τ-transition. The composition $X \oplus S$ can also reach such a state, as T is a subgraph of S that only differs in t_1 and s_1. □

4 Maximal Controller

Operating guidelines $OG(S)$ describe the set of all controllers of a service S, represented by an annotated automaton. In this section we represent operating guidelines of S by a single controller. This controller is the *maximal controller* $mc(S)$ [21], as it is larger in the accordance preorder than all controllers R of S.

Definition 1 ([21]). *A* maximal controller *of a controllable service S is a service $mc(S)$ such that:* $(\forall R :: \; R \in Controllers(S) \;\Leftrightarrow\; R \leq mc(S))$.

Using the definitions of $\leq$ and $Controllers(S)$, we can simplify the two implications inside the equivalence $\Leftrightarrow$:



(a) Rule 1 (b) Rule 2

Fig. 3. Two accordance-preserving transformation rules

$\Leftarrow$: $mc(S)$ is a controller: $mc(S) \in \mathit{Controllers}(S)$, and
$\Rightarrow$: $mc(S)$ is larger than all controllers: $(\forall R :: R \in \mathit{Controllers}(S) \Rightarrow R \leq mc(S))$.

The maximal controller is unique up to accordance, and in Sect. 5 we show how to construct one. In the following, we study how a maximal controller behaves with respect to the accordance preorder. In [21] we derived a Galois connection (see [3]) for the maximal controller and the accordance preorder.

Proposition 4 ([21]). *For each two controllable services R and S:*

$$R \leq mc(S) \;\Leftrightarrow\; S \leq mc(R)$$

Given such a Galois connection and the preorder $\leq$, we obtain all kinds of standard properties similar to those mentioned in [21]. Moreover, we can use Definition 1 to prove a stronger version of two properties.

Lemma 1. *For each two controllable services S and T:*

$$T \leq S \;\Leftrightarrow\; mc(S) \leq mc(T) \quad \textit{and} \quad S =_{acc} mc(mc(S))$$

Proof. In succession, we calculate:

$mc(S) \leq mc(T)$
$\Leftrightarrow$ {indirect inequality}
$(\forall R :: R \leq mc(S) \Rightarrow R \leq mc(T))$
$\Leftrightarrow$ {Definition 1 (twice); set theory}
$\mathit{Controllers}(S) \subseteq \mathit{Controllers}(T)$
$\Leftrightarrow$ {definition of $\leq$}
$T \leq S$

$S =_{acc} mc(mc(S))$
$\Leftrightarrow$ {indirect equality}
$(\forall T :: T \leq S \Leftrightarrow T \leq mc(mc(S)))$
$\Leftrightarrow$ {Proposition 4}
$(\forall T :: T \leq S \Leftrightarrow mc(S) \leq mc(T)))$
$\Leftrightarrow$ {first part of this lemma}
true

Consequently, the lemma holds. □

These two properties are illustrated in Fig. 4, and they turn out to be useful in further proofs and applications.

5 Finite Maximal Controllers

In this section, we provide a construction of a maximal controller as a finite service, and we relate it to some existing results on operating guidelines.

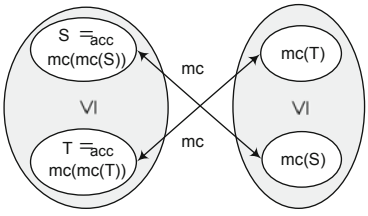


Fig. 4. Effect of mc on $\leq$: the maximal controller turns the accordance preorder upside-down, and the maximal controller of the maximal controller of a service is in the same equivalence class as the original service

5.1 Construction

For a given service S, there are several maximal controllers $mc(S)$, like there are several most-permissive controllers. In the following we construct a particular *finite maximal controller*, denoted by $M(S)$. In [21] we informally conjectured the computation of $M(S)$ from finite operating guidelines $OG(S) = (Z_S, \phi)$: "$M(S)$ is obtained from the operating guidelines by replacing every labeled state q of Z_S by a nondeterministic internal choice between all the valid combinations of outgoing edges from this state." A combination of outgoing edges (and a final proposition) of a state q is valid if it satisfies the annotation $\phi(q)$.

Definition 2 (Construction of $M(S)$). *Let S be a controllable service, let (Z_S, ϕ) be operating guidelines of S, and let $Z_S = (Q, q_0, I, O, \delta)$. Service $M(S) = (Q', q_0', \Omega', I, O, \delta')$ is defined as:*

- $Q' = Q \cup \{q_\beta \mid q \in Q \ \wedge \ \beta \models \phi(q)\}; \quad q_0' = q_0; \quad \Omega' = \{q_\beta \mid \beta(\mathit{final})\};$
- $\delta' = \{(q, \tau, q_\beta) \mid q \in Q \ \wedge \ \beta \models \phi(q)\} \ \cup \ \{(q_\beta, m, q') \mid (q, m, q') \in \delta \ \wedge \ \beta(m)\}.$

The upper part of Fig. 5 sketches the construction of a fragment of $M(S)$ from operating guidelines $OG(S)$. For every state q in Z_S and assignment β that satisfies $\phi(q)$, there is a state q_β, a τ-transition from q to q_β, and, for each m in β, a transition from q_β to the corresponding successor q'. If β assigns *true* to *final* at q, then q_β is a final state.

Given the direct relation between $OG(S)$ and $M(S)$, algorithms for $OG(S)$ can easily be adapted to $M(S)$. For simplicity reasons, our construction of $M(S)$ is based on all satisfying assignments β of each formula $\phi(q)$. Thus, in worst case, the size of $M(S)$ is proportional to the size of $OG(S)$ multiplied by 2 to the power of the number of interface channels (which we consider to be static). As the size of operating guidelines is exponential [30] in the size of the size of automaton S, this static factor is negligible.

For applications that rely on maximal controllers, we sketch how the size of $M(S)$ can be reduced. Consider operating guidelines with formulae $\phi(q)$ in *disjunctive normal form* (i. e., a disjunction of conjunctions). Another finite maximal controller can be obtained by replacing every labeled node q by a nondeterministic internal choice between all the disjuncts d in $\phi(q)$, where each disjunct d is a conjunction of outgoing edges (or the proposition *final*). In addition, this

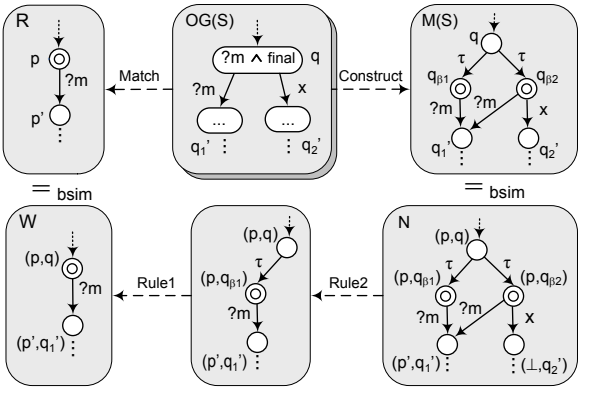


Fig. 5. A service R that matches with $OG(S)$ can be obtained from $M(S)$. First replace R by a bisimilar W, and replace $M(S)$ by a bisimilar N. Then transform N into W using Rule 1 and Rule 2.

choice is extended with an external choice between the outgoing edges that do not occur in $\phi(q)$. The accordance equivalence of the two finite maximal controllers can be proved in the proof style of Sect. 3 using that for every disjunct d there is a satisfying assignment β that matches exactly, and for every satisfying assignment β (a conjunction) there is a disjunct d that is satisfied by (a subset of) β.

5.2 Validity

In [21] it was conjectured that the construction from Definition 2 yields a maximal controller. In the remainder of this section, in particular in Theorem 2, we prove that $M(S)$ is indeed a maximal controller of service S. To this end, we show in Lemma 2 that $M(S)$ is a controller of S, and in Lemma 3 that $M(S)$ is larger than every controller of S.

Lemma 2. *For every controllable service S: $M(S) \in Controllers(S)$.*

Proof. By Proposition 1: $Controllers(S) = Match(OG(S))$. Let $OG(S) = (Z_S, \phi)$ be operating guidelines of S with $Z_S = (Q, q_0, I, O, \delta)$, and $M(S)$ be constructed from (Z_S, ϕ) as described in Definition 2. The required simulation relation ϱ is: $\varrho = \{(q, q) \mid q \in Q\} \cup \{(q_\beta, q) \mid q \in Q \wedge \beta \models \phi(q)\}$ using that every state q in (Z_S, ϕ) has a τ-loop, and $\{\tau\}$ satisfies $\phi(q)$ holds. □

In the next lemma we prove that $M(S)$ is larger than every controller of S.

Lemma 3. *For every controllable service S:*

$$(\forall R :: \; R \in Controllers(S) \;\Rightarrow\; R \leq M(S))$$

Proof. Let S be a controllable service and R be a service. After applying Proposition 1, let (Z_S, ϕ) be operating guidelines of S with $Z_S = (Q, q_0, I, O, \delta)$. We

first replace R and $M(S)$ by bisimilar service automata W and N that have similar states:

- Let W be the following service:
  - $Q_W = Q_R \times Q$; $q_{0W} = (q_{0R}, q_0)$; $\Omega_W = \Omega_R \times Q$; $I_W = I_R$; $O_W = O_R$;
  - $\delta_W = \{((q,r), m, (q',r')) \mid (q,m,q') \in \delta_R \ \wedge \ (r,m,r') \in \delta\}$

  Service W synchronizes R with Z_S, but ignores transitions that are not shared. Each state (q,r) from W is bisimilar to q from R, as Z_S simulates R, and Z_S is deterministic. The simulation relation between the reachable part of W and (Z_S, ϕ) is $\varrho = \{((q,r), r) \mid (q,r) \in Q_W\}$.
- Let N be the service $(Q'', q_0'', \Omega'', I, O, \delta'')$ defined as:
  - $Q'' = (Q_R \cup \{\bot\}) \times (Q \ \cup \ \{q_\beta \mid q \in Q \ \wedge \ \beta \models \phi(q)\})$;
  - $q_0'' = (q_{0R}, q_0)$; $\quad \Omega'' = (Q \cup \{\bot\}) \times \{q_\beta \mid \beta(\mathit{final})\}$
  - $\delta'' = \ \{((p,q), \tau, (p, q_\beta)) \mid q \in Q \ \wedge \ \beta \models \phi(q)\} \ \cup$
    $\{((p,q_\beta), m, (p',q')) \mid (q,m,q') \in \delta \ \wedge \ \beta(m) \ \wedge$
    $((p,m,p') \in \delta_R \ \vee \ (p' = \bot \ \wedge \ \neg(\exists r :: \ (p,m,r) \in \delta_R)))\}$

  Service N synchronizes $M(S)$ with R, and uses states $(\bot, q)$ for transitions that are not in R. Each state (p,q) in N is bisimilar to state q in $M(S)$.

The states of W are a subset of the states of N; see Fig. 5. To prove $W \leq N$, we show how to obtain W from N using Rule 1 and Rule 2.

Consider every reachable state in W. Each state (q_R, q) in W simulates state q in Z_S and satisfies $\phi(q)$. By construction, the state (q_R, q) in N offers a nondeterministic internal choice between all assignments β that satisfy $\phi(q)$. Using Rule 2, we can remove from state (q_R, q) in N all τ-branches except the one leading to the state (q_R, q_β) that has the same outgoing edges (and final proposition) as state (q_R, q) in W. Using Rule 1, we can eliminate this τ-edge. By construction, the remaining state has the same direct successors as (q_R, q) in W. □

Theorem 2. *For every controllable service S, the service $M(S)$ is a maximal controller of S, i. e., it is a solution for $mc(S)$ in Definition 7.*

Proof. Follows from Lemma 2 and Lemma 3. □

5.3 Canonicity

Accordance equivalent services may have different operating guidelines, and hence Definition 2 may give maximal controllers that are not isomorphic. We show that the resulting maximal controllers are bisimilar.

Lemma 4. *For any two controllable services S and T:*

$$S =_{acc} T \quad \Rightarrow \quad M(S) =_{bsim} M(T)$$

Proof. Let S and T be services such that $S =_{acc} T$. By definition of $=_{acc}$, the operating guidelines of S and T have the same sets of matching services. Using Proposition 2 this means that the automata underlying the operating guidelines simulate each other, and the annotations of the states imply each other. As the automata are deterministic, they are bisimilar and related states have equivalent annotations. From Definition 2, we conclude $M(S) =_{bsim} M(T)$. □

6 Applications to Service Replaceability

In this section we illustrate three applications of a maximal controller in the context of service replaceability. The main idea is that, as a maximal controller is a service rather than operating guidelines, we can apply service operations to it, including composition, operating guidelines computation, and maximal controller computation. These results extend the results in [26] that are published in nonrefereed workshop proceedings.

6.1 Deciding Replaceability

The first application is service replaceability, which addresses the question whether a service T can replace a service S. Accordance guarantees this independently of the context of S, i. e., by ensuring that every controller of S is a controller of T.

The sets of controllers are infinite, but operating guidelines are finite characterizations of these sets. The current procedure [27] for deciding accordance requires the computation of operating guidelines of S, operating guidelines of T, and the verification of a refinement relation between them; see Proposition 2. Using a maximal controller $mc(S)$, we can decide accordance by checking deadlock freedom in the composition of T and $mc(S)$.

Theorem 3. *For each service T and controllable service S:*

$$T \leq S \quad \Leftrightarrow \quad DF(mc(S) \oplus T)$$

Proof. $DF(mc(S) \oplus T)$ iff $mc(S) \in Controllers(T)$ (by definition of controller) iff $mc(S) \leq mc(T)$ (by Definition 1) iff $T \leq S$ (by Lemma 1) □

This decision procedure requires the computation of one maximal controller and one check for deadlock freedom. In contrast to [27], we have not yet implemented this procedure, but we expect that this procedure is also feasible in practice.

6.2 Characterizing all Replaceable Services

Our main application of maximal controllers is the computation of operating guidelines that represent the set $Accord(S)$ of all services that can replace S. As a maximal controller of S represents all possible controllers of S, we can represent $Accord(S)$ by operating guidelines of $mc(S)$.

Theorem 4. *For each controllable service S: $Accord(S) = Controllers(mc(S))$.*

Proof. Follows from Theorem 3 and the definition of *Accord* and *Controllers*. □

Using Theorem 4 and Proposition 1, we can compute $Accord(S)$ as operating guidelines $OG(mc(S))$. The matching algorithm enables us to select from $OG(mc(S))$ any service T that can replace S. Each T can be seen as a communication skeleton (or an abstract BPEL process) that can be refined, e. g.,

using the accordance-preserving rules from [1,14]. Our approach has beneficial practical implications. Using the set $Accord(S)$ we can find all services that can replace S, whereas this is not possible applying existing transformation rules. This may potentially save development time when creating a service.

Example 4. Figure 2 depicts $OG(mc(B))$, representing all services that can replace service B. This set contains services A, B, and C in Fig. 1.

We can also apply all existing techniques for operating guidelines. Suppose we want to impose additional requirements on the service T. Then we can restrict $OG(mc(S))$ to services that satisfy certain behavioral constraints [18], or that can perform certain activities [28]. If service T must be similar (in terms of an edit distance) to another given service T' (which may not accord with S), we can use the approach of [17] to compute from $OG(mc(S))$ such a service T together with the edit actions for transforming T' into T.

Finally, $OG(mc(S))$ provides another way of deciding accordance: T accords with S if T matches with $OG(mc(S))$. If $OG(mc(S))$ is not given, then it is unlikely that this procedure improves on the one from Sect. 6.1. However, if $OG(mc(S))$ has already been computed, then we expect this procedure to be practically feasible. It is further work to confirm this using real experiments.

6.3 Constructing a Public View of a Service

In the context of inter-organizational processes, service providers need to publish information about the services they offer. On the one hand, they have to provide enough details to correctly interact with the services, while on the other hand hiding all other details of the service. A popular approach is to publish a *public view* (or interaction protocol) of the service [2,16].

Using the maximal controller, we can construct a canonical public view of a service. A maximal controller encodes the set of all controllers: Lemma 1 shows that equivalent services $mc(S)$ and $mc(T)$ indicate that services S and T have the same controllers. By applying Lemma 1 twice, we can also conclude that equivalent services $mc(mc(S))$ and $mc(mc(T))$ indicate that services S and T have the same controllers.

As Lemma 1 indicates that service $mc(mc(S))$ is accordance equivalent to service S, a service provider that offers service S could safely publish $mc(mc(S))$ instead. Although it is likely that $OG(S)$ has less states than $mc(mc(S))$, publishing $mc(mc(S))$ instead of $OG(S)$ has two benefits: $mc(mc(S))$ is a single service rather than a representation of a set of services, and it represents the offered service rather than its controllers.

Using the specific maximal controller as described in Definition 2, $M(M(S))$ is a canonical element of the equivalence class of S.

Theorem 5. *For any two controllable services S and T:*

$$S =_{acc} T \quad \Leftrightarrow \quad M(M(S)) =_{bsim} M(M(T))$$

Proof. We justify the equivalence by proving two implications:
$\Rightarrow$: Follows from Lemma 4 (twice), as bisimulation implies accordance.
$\Leftarrow$: Follows from Lemma 1 (twice), as bisimulation implies accordance. □

With this result, we can show that $M(S)$ can be used as a canonical representation of the equivalence class of S.

Theorem 6. *For any two controllable services S and T:*

$$S =_{acc} T \quad \Leftrightarrow \quad M(S) =_{bsim} M(T)$$

Proof. We justify the equivalence by proving two implications:
$\Rightarrow$: See Lemma 4.
$\Leftarrow$: Using Lemma 4 we conclude $M(M(S)) =_{bsim} M(M(T))$, as bisimulation implies accordance. Using Theorem 5 we then obtain $S =_{acc} T$. □

7 Related Work

Apart from the accordance preorder, several other preorders have been proposed to characterize and to decide service replaceability, see [29,11,4,15,6,27,5], for instance. The accordance preorder coincides with the stable failures preorder [13]. Closest to accordance is the subcontract preorder [15], which coincides for τ-free services with accordance. For a more detailed comparison of accordance with other preorders, we refer to [27].

In Theorem 3, we showed how the notion of a maximal controller can be used to decide accordance and hence service replaceability. Similar decision procedures for different preorders have been studied in [7,9]. Moreover, the notion of a maximal controller is related to the notion of a canonical dual from [8]. A trivial construction method in their (restricted) setting is proposed in [8], but this method does not apply in our setting.

The notion of a public view of a service has been considered in [2,16,10], for instance. In contrast to the construction algorithms in [2,10], our proposed public view based on a maximal controller is a canonical construction that is independent of reduction rules while acting as a "service obfuscator". Another approach is to publish a representation of all controllers of S, e. g., using operating guidelines [19]. Operating guidelines describe the communication structure that is necessary for deadlock-free interaction. The advantage of publishing operating guidelines is that matching a single service R with operating guidelines $OG(S)$ of a service S is in general less complex than model checking the composition of R and the public view of S for deadlock freedom.

8 Conclusion and Further Work

Operating guidelines have been proposed as a finite representation of the infinite set of controller services that can interact properly with a given service. They have already been used for many applications in the context of service-oriented systems, yet no operating guidelines could be computed to represent all services that can replace a given service.

In this paper, we have broadened the applicability of operating guidelines by adding the notion of a maximal controller $mc(S)$ of a service S. A maximal controller $mc(S)$ is a largest controller of S that encodes all controllers of S. We have shown a way to construct a maximal controller, and proved its correctness.

A maximal controller $mc(S)$ allows for at least three applications. First, operating guidelines of $mc(S)$ represent the set of services that can replace service S, and they can be further manipulated using various existing techniques for operating guidelines. Second, using $mc(S)$ we can also construct a public view of a service, and third decide whether a service T can replace service S.

Operating guidelines and maximal controllers are related, as they can be seen as different encodings of the set of controllers. On the one hand, operating guidelines have less states and are more suitable for human comprehension. On the other hand, a maximal controller is a single, albeit larger, service that is more suitable as input for further operations on services.

It is further work to study maximal controllers for other correctness notions than deadlock freedom (see [12] for some initial partial results), and for other sets of services than operating guidelines.

Acknowledgements. Authors Mooij and Voorhoeve participate in the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

References

1. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From public views to private views – correctness-by-design for services. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 139–153. Springer, Heidelberg (2008)
2. van der Aalst, W.M.P., Weske, M.: The P2P approach to interorganizational workflows. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 140–156. Springer, Heidelberg (2001)
3. Backhouse, R.: Galois connections and fixed point calculus. In: Blackhouse, R., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 89–148. Springer, Heidelberg (2002)
4. Benatallah, B., Casati, F., Toumani, F.: Representing, Analysing and Managing Web Service Protocols. Data Knowl. Eng. 58(3), 327–357 (2006)
5. Bonchi, F., Brogi, A., Corfini, S., Gadducci, F.: A net-based approach to web services publication and replaceability. Fundam. Inform. 94(3-4), 305–330 (2009)
6. Bravetti, M., Tennenholtz, M.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)
7. Brinksma, E.: A theory for the derivation of tests. In: Protocol Specification, Testing, and Verification VIII, pp. 63–74. North-Holland, Amsterdam (1988)
8. Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: PPDP 2009, pp. 219–230. ACM, New York (2009)
9. Dill, D.L.: Trace theory for automatic hierarchical verification of speed-independent circuits. MIT Press, Cambridge (1989)

10. Eshuis, R., Grefen, P.W.P.J.: Composing services into structured processes. Int. J. Cooperative Inf. Syst. 18(2), 309–337 (2009)
11. Fournet, C., Hoare, S.T., Rajamani, S.K., Rehof, J.: Stuck-free conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004)
12. van Hee, K., Mooij, A.J., Sidorova, N., van der Werf, J.M.: Soundness-preserving refinements of service compositions. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 131–145. Springer, Heidelberg (2010)
13. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall International series in computing science. Prentice-Hall, Englewood Cliffs (1985)
14. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract WS-BPEL processes. In: WWW 2008, pp. 785–794. ACM, New York (2008)
15. Laneve, C., Padovani, L.: The must preorder revisited. In: Caires, L., Li, L. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 212–225. Springer, Heidelberg (2007)
16. Leymann, F., Roller, D., Schmidt, M.: Web services and business process management. IBM Systems Journal 41(2), 198–211 (2002)
17. Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 132–147. Springer, Heidelberg (2008)
18. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 271–287. Springer, Heidelberg (2007)
19. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
20. Milner, R.: Communication and concurrency. Prentice-Hall, Englewood Cliffs (1989)
21. Mooij, A.J., Voorhoeve, M.: Proof techniques for adapter generation. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 207–223. Springer, Heidelberg (2009)
22. Mooij, A.J., Voorhoeve, M.: Trading off concurrency to generate behavioral adapters. In: ACSD 2009, pp. 109–118. IEEE, Los Alamitos (2009)
23. Murata, T.: Petri nets: Properties, analysis and applications. Proc. of the IEEE 77(4), 541–580 (1989)
24. Papazoglou, M.P.: The challenges of service evolution. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 1–15. Springer, Heidelberg (2008)
25. Park, D.: Concurrency and automata on infinite sequences. In: Proc. of the 5th GI-Conference on Theoretical Computer Science, pp. 167–183. Springer, Heidelberg (1981)
26. Parnjai, J., Stahl, C., Wolf, K.: A finite representation of all substitutable services and its applications. In: ZEUS 2009, vol. 438, pp. 8–14. CEUR (March 2009)
27. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding Substitutability of Services with Operating Guidelines. ToPNoC II 2(5460), 172–191 (2009)
28. Stahl, C., Wolf, K.: Covering places and transitions in open nets. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 116–131. Springer, Heidelberg (2008)
29. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)
30. Wolf, K.: Does my service have partners? ToPNoC 5460(II), 152–171 (2009); special Issue on Concurrency in Process-Aware Information Systems

Soundness-Preserving Refinements of Service Compositions

Kees M. van Hee, Arjan J. Mooij, Natalia Sidorova, and Jan Martijn van der Werf

Department of Mathematics and Computer Science*,
Technische Universiteit Eindhoven, The Netherlands
{K.M.v.Hee,A.J.Mooij,N.Sidorova,J.M.E.M.v.d.Werf}@tue.nl

Abstract. Soundness is one of the well-studied properties of processes; it denotes that a final state can be reached from every state that is reachable from the initial state. Soundness-preserving refinements are important for enabling the compositional design of systems.

In this paper we concentrate on refinements of service compositions. We model service compositions using Petri nets, and consider specific pairs of places that belong to different services. Starting from a sound service composition, we show how to check whether such a pair of places can be refined by another sound service composition, so that soundness is preserved through the refinement.

Keywords: Service composition, refinement, Petri net, soundness, verification.

1 Introduction

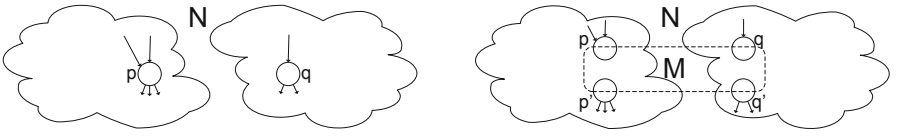
Recent developments such as component-based software engineering (CBSE) and service-oriented architectures (SOA) have led to systems that are composed from many services. Each service delivers a specific functionality, and communicates asynchronously with some other services using messages. In turn, a service itself may be composed out of several other (communicating) services, resulting in an intricate network of services.

This trend became only more visible with the adoption of the Software as a Service (SaaS) paradigm that facilitates the communication across boundaries of organizations. As a consequence, it became virtually impossible for a single organization to obtain a full model of the system, and hence it became even more challenging to ensure its (behavioral) correctness.

In this paper we study compositional design methods that ensure correctness of service compositions based on properties of communicating pairs of services. One of the main formalisms for modeling and analyzing systems that communicate asynchronously are Petri nets, which we use in this paper. The behavioral

* Author Mooij participates in the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 131–145, 2011.



(a) Net N (b) Net $N \odot M$

Fig. 1. Refinement of (synchronizable) places

correctness property that we consider is *soundness* [1], or *weak termination*, which requires that a final state can be reached from every state that is reachable from the initial state. Soundness has been studied extensively, and has proved to be practically relevant.

Compositional techniques for design and analysis have a long tradition. A nice overview of fundamental refinement techniques for Petri nets is given in [4,29]. In the context of component-based systems and service-oriented technologies, a lot of research considers communicating systems, see e.g., [14,26,27], in combination with some variants of soundness. However, these works focus on "horizontal" modularization (i.e., composition) of communicating systems, while we are interested here in "vertical" modularization (i.e., refinement). Conditions for vertical modularization were given by [28,12], regarding soundness-preserving refinements of a single place (in a Petri net).

In this paper we consider the refinement of a pair of places p and q in a Petri net N by a sound workflow net M with two designated initial (source) places p and q and two final (sink) places p' and q'; see Fig. 1. Both net N and net M model service compositions that may involve multiple communicating services. We define conditions for refined net N and refining net M in isolation such that the refinement is sound.

Overview. In Section 2, we summarize the basic definitions related to Petri nets and the accordance relation. In Section 3, we introduce the refinement of synchronizable places and give the intuition behind this concept. In Section 4, we present a homogeneous criterion for refinement based on two soundness checks. In Section 5, we formally prove its correctness. We conclude in Section 6 with some conclusions and further work.

2 Preliminaries

Let S be a set. A *sequence* σ of length $n \in \mathbb{N}$ over S is a function $\sigma : \{1, \ldots, n\} \to S$; we denote its length by $|\sigma| = n$. If $|\sigma| = 0$, then σ is the *empty sequence* ϵ. The set of all finite sequences over S is denoted by S^*.

A *bag* (or *multiset*) m over S is a function $m : S \to \mathbb{N}$. We use '.' to denote function application; so, for $s \in S$, $m.s$ denotes the number of occurrences of s in m. We write $\mathbb{N}^S$ for the set of all bags over S, and $[s]$ for the bag containing one occurrence of $s \in S$. We use $+$ for the sum of two bags, and $\leq$ for the comparison of two bags. Sets can be seen as bags in which all elements have multiplicity one.

Let $\mathcal{A}$ be the universe of actions, not including silent (or internal) action τ.

Definition 1 (Labeled transition system). *A* labeled transition system, *LTS for short, L is a 4-tuple $(S, \longrightarrow, s_i, \Omega)$, where S is a set of* states*; $\longrightarrow \subseteq S \times (\mathcal{A} \cup \{\tau\}) \times S$ is a* transition relation*; $s_i \in S$ is the* initial state*; and $\Omega \subseteq S$ is the set of* final states*.*

For $s, s' \in S$ and $a \in \mathcal{A}$, we write $s \xrightarrow{a} s'$ if and only if $(s, a, s') \in \longrightarrow$. A state $s \in S$ is called a *deadlock* if no action $a \in \mathcal{A} \cup \{\tau\}$ and state $s' \in S$ exist such that $s \xrightarrow{a} s'$. If for some $\sigma \in (\mathcal{A} \cup \{\tau\})^*$ of length n, and states $s_i \in S$ for $0 \leq i \leq n$ such that $s_{i-1} \xrightarrow{\sigma.i} s_i$ for $0 < i \leq n$, we write $s_0 \xrightarrow{\sigma} s_n$. The set of reachable states from a given state $s \in S$ is defined as $\mathcal{R}(L, s) = \{s' \in S \mid \exists \sigma \in (\mathcal{A} \cup \{\tau\})^* : s \xrightarrow{\sigma} s'\}$.

Definition 2 (Weak termination). *An LTS L is* weakly terminating *if for every state $s \in \mathcal{R}(L, s_i)$, $\Omega \cap \mathcal{R}(L, s) \neq \emptyset$ holds.*

2.1 Petri Nets, Workflows and Soundness

A *Petri net* is a 3-tuple $N = (P, T, F)$, where P and T are two disjoint sets of *places* and *transitions* respectively; and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. The elements from the set $P \cup T$ are called the *nodes* of N. Elements of F are called *arcs*. Given a node $n \in (P \cup T)$, we define its *preset* ${}^\bullet n = \{n' \mid (n', n) \in F\}$, and its *postset* $n^\bullet = \{n' \mid (n, n') \in F\}$. Graphically, places are depicted as circles, transitions as squares, and arcs as arrows.

Markings are the states of a net; each *marking* m of a Petri net $N = (P, T, F)$ is a bag over P. A transition t is *enabled* in a marking m if ${}^\bullet t \leq m$; *firing* an enabled transition t in marking m yields a marking m' such that $m' + {}^\bullet t = m + t^\bullet$.

A *system* is a triple (N, m, Ω), where N is a Petri net, $m \in \mathbb{N}^P$ is the initial marking and $\Omega \subseteq \mathbb{N}^P$ is a set of final markings. The semantics of a system (N, m, Ω) is defined as an LTS $(\mathbb{N}^P, \longrightarrow, m, \Omega)$, where $(m, t, m') \in \longrightarrow$ if and only if $m' + {}^\bullet t = m + t^\bullet$ and ${}^\bullet t \leq m$.

Workflow nets are special Petri nets with one initial place and one final place.

Definition 3 (Workflow net, soundness). *A Petri net (P, T, F) is called a* workflow net *if (1) there exists exactly one place $i \in P$, called the* initial place*, such that ${}^\bullet i = \emptyset$, (2) there exists exactly one place $f \in P$, called the* final place*, such that $f^\bullet = \emptyset$, and (3) all nodes are on a path from i to f.*

A workflow net N is sound *if the LTS semantics of system $(N, [i], \{[f]\})$ is weakly terminating.*

In the temporal logic CTL (Computation Tree Logic, [9]), weak termination (and hence soundness) can be expressed using the "AG EF" pattern, where AG refers to every reachable state, and EF refers to the existence of a (terminating) path. Such properties can be checked for Petri nets using tools like LoLA [25].

2.2 Open Nets and Composition

We use an extension of Petri nets that is called open nets [13,18]. To model external asynchronous communication, open nets have an interface that consists of input places and output places.

Definition 4 (Open (Petri) net, soundness). *An* open (Petri) net *is a* 7*-tuple* $(P, T, F, P_i, P_o, m_0, \Omega)$*, where* $((P, T, F), m_0, \Omega)$ *is a system;* $P_i \subseteq P$ *is a set of input places such that* ${}^\bullet p = \emptyset$ *for all* $p \in P_i$*;* $P_o \subseteq P$ *is a set of output places such that* $p^\bullet = \emptyset$ *for all* $p \in P_o$*; and* $m.p = 0$ *for all markings* $m \in \Omega \cup \{m_0\}$ *and places* $p \in P_i \cup P_o$*.*

A closed net *is an open net without asynchronous interface places, i.e.,* $P_i = P_o = \emptyset$*. A closed net* N *is called* sound*, denoted by SD.*N*, if the LTS semantics of its system is weakly terminating.*

To model synchronous communication, we extend open nets as in [22] with a total labeling function L, which assigns to every transition a label that denotes the synchronous port (if any) it is connected to. If a transition is not connected to any synchronous port, it is assigned the auxiliary label τ. A closed net also has no synchronous interface ports, i.e., $(\forall t : t \in T : \ L.t = \tau)$.

Traditional open nets (without synchronous ports) can be composed by fusing interface places that are an input place of one net and an output place of the other, resulting in internal places. Two nets are composable if and only if their shared interface places are of this kind. Open nets with synchronous ports can be composed by also fusing each pair of transitions from the two nets with identical non-τ labels, resulting in τ-labeled transitions.

Without loss of generality, we assume that all nodes except the interfaces of the involved nets are disjoint, which can be achieved by renaming the internal (non-interface) places and transitions.

Definition 5 (Composition, [22]). *Let* N_1 *and* N_2 *be open nets.* N_1 *and* N_2 *are* composable *iff* $P_{i1} \cap P_{i2} = \emptyset$ *and* $P_{o1} \cap P_{o2} = \emptyset$*. If* N_1 *and* N_2 *are composable, their* composition $N = N_1 \oplus N_2$ *is defined by*

$$P = P_1 \cup P_2; \ \ P_i = (P_{i1} \cup P_{i2}) \setminus (P_{o1} \cup P_{o2}); \ \ P_o = (P_{o1} \cup P_{o2}) \setminus (P_{i1} \cup P_{i2});$$
$$\Omega = \{m_1 + m_2 \mid m_1, m_2 : \ m_1 \in \Omega_1 \ \wedge \ m_2 \in \Omega_2\}; \quad m_0 = m_{01} + m_{02};$$

$$T = T_f \cup T_s;$$
$$T_f = \{t \mid t : \ t \in T_1 \ \wedge \ (L_1.t = \tau \ \vee \ (\forall t' : t' \in T_2 : L_1.t \neq L_2.t'))\}$$
$$\cup \ \{t \mid t : \ t \in T_2 \ \wedge \ (L_2.t = \tau \ \vee \ (\forall t' : t' \in T_1 : L_2.t \neq L_1.t'))\};$$
$$T_s = \{\{t_1, t_2\} \mid t_1, t_2 : \ t_1 \in T_1 \ \wedge \ t_2 \in T_2 \ \wedge \ L_1.t_1 = L_2.t_2 \ \wedge \ L_1.t_1 \neq \tau\};$$

$$L = \{[t, (L_1 \cup L_2).t] \mid t : \ t \in T_f\} \ \cup \ \{[\{t_1, t_2\}, \tau] \mid t_1, t_2 : \ \{t_1, t_2\} \in T_s\};$$

$$F = ((F_1 \cup F_2) \ \cap \ ((P \cup T_f) \times (P \cup T_f)))$$
$$\cup \ \{[p, \{t_1, t_2\}] \mid p, t_1, t_2 : \ \{t_1, t_2\} \in T_s \ \wedge \ ([p, t_1] \in F_1 \ \vee \ [p, t_2] \in F_2)\}$$
$$\cup \ \{[\{t_1, t_2\}, p] \mid p, t_1, t_2 : \ \{t_1, t_2\} \in T_s \ \wedge \ ([t_1, p] \in F_1 \ \vee \ [t_2, p] \in F_2)\}.$$

2.3 Accordance

A *controller* of an open net S is an open net R such that $SD.(R \oplus S)$ holds. A *controllable* open net is an open net that has at least one controller.

Definition 6 (Accordance pre-order). *The accordance pre-order $\leq$ on open nets S and T is defined as:* $S \leq T \;\equiv\; (\forall R ::\; SD.(R \oplus T) \;\Rightarrow\; SD.(R \oplus S))$.

The accordance pre-order [3] is equivalent to the conflict pre-order from [17], and the sub-contract pre-order from [5]. In [20] the relation between this pre-order and other pre-orders (in particular, fair testing [24]) has been studied.

Definition 7 (Maximal controller, [21]). *A maximal controller of a controllable open net S is an open net $mc.S$ such that:*

$$(\forall R ::\; SD.(R \oplus S) \;\equiv\; R \leq mc.S)$$

We can simplify the two implications inside the equivalence $\equiv$ as follows:

$\Leftarrow$: $mc.S$ is a controller: $SD.(mc.S \oplus S)$, and
$\Rightarrow$: $mc.S$ is larger than all controllers: $(\forall R ::\; SD.(R \oplus S) \;\Rightarrow\; R \leq mc.S)$.

Using the maximal controller, we can decide accordance by checking deadlock-freedom in the composition of T and a maximal controller of S. Similar decision procedures for different pre-orders have been studied in [6,10].

Proposition 1 (Deciding accordance using maximal controller, [19]). *For each open net S and controllable open net T:* $S \leq T \;\equiv\; SD.(S \oplus mc.T)$.

3 Synchronizable Places

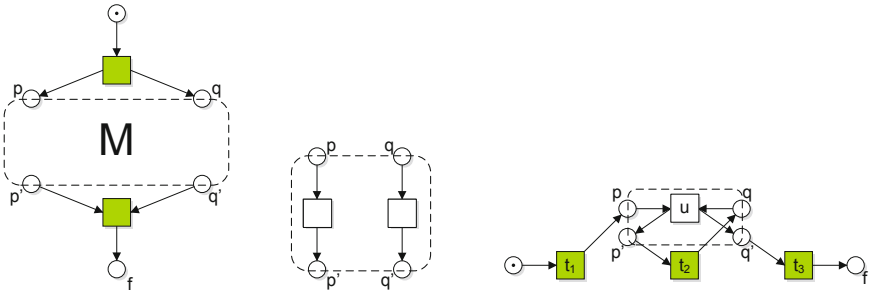
In this section we explore the problem of preserving soundness while refining pairs of (synchronizable) places. We apply a semi-formal style.

3.1 Introduction

Service compositions consist of several independent services that communicate via interfaces. We use Petri nets to model the communication between such services. Our aim is to support the design of nets in a top-down manner, in particular by refining pairs of places.

Suppose a closed net N is given, which contains two internal places p and q (which do not occur in the initial marking nor in any final marking). These two places can be refined by an open net M with input places p and q, and output places p' and q'. This refinement, denoted by $N \odot M$, is sketched in Fig. 1.

If places p and q in N are related to two different services, we thus impose the additional communication protocol M on the two services to which places p and q in N belong. We assume that open net M also models several services, i.e., input place p models the initial state of one service, and output place p'



(a) Workflow (b) Forward (c) Sequence

Fig. 2. Exploration: some (counter) examples

models the end state of the service; similarly for q and q' in relation to another service. Moreover, net M consumes a token from p before it produces a token in p'; similarly for places q and q'. Finally, we assume that the net in Fig. 2(a), which contains M, is a sound workflow.

In what follows, we study under which conditions p and q in net N can be called *synchronizable*, i.e., under which conditions the refinement $N \odot M$ is sound. Such conditions must implicitly approximate M, but independently of any specific M.

3.2 Soundness

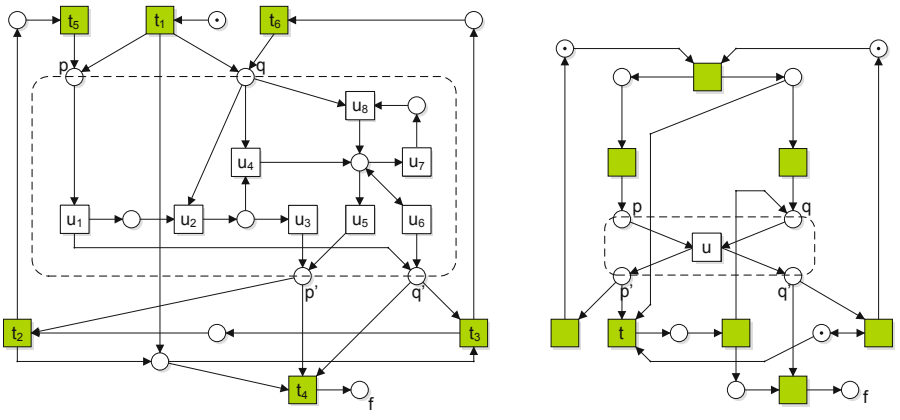
Soundness of N is a necessary condition for the refinement. Consider for example the net M in Fig. 2(b), for which $N \odot M$ is equivalent (fusion of series places [23]) to N, for every net N. For $N \odot M$ to be sound, net N must be sound.

However, this is not a sufficient condition. An example refinement $N \odot M$ is depicted in Fig. 2(c). Net N contains places p and q that are "sequentially connected" by transition t_2, and net M is a simple synchronizing net. Although N is sound, $N \odot M$ reaches a deadlock after firing transition t_1. From this example we conclude that it should be possible to mark places p and q in net N at the same time.

3.3 Refinement with the Simplest Synchronizing Net

The previous subsection suggests to check soundness of N refined with the simplest synchronizing net M, viz., a single transition, as depicted in Fig. 2(c).

However, this is not a sufficient condition. An example refinement $N \odot M$ is depicted in Fig. 3(a). Net N fires transition t_1, followed by an alternation between transitions t_3, t_6 and t_2, t_5, and finally transition t_4. Net N is sound, even after refinement with the simplest synchronizing net. However, in the refinement $N \odot M$, net M can consume a second token from q before producing a token in p' (or net N can produce a second token in q before consuming a token from p'). After firing transitions t_1, u_1 and u_2, transitions t_3 and t_6 can fire, and



(a) Deadlock after refinement (b) Livelock after refinement

Fig. 3. Exploration: more (counter) examples

hence transitions u_4 and u_7 can fire; thus resulting in a deadlock. The net is even unbounded, as after transitions t_1, u_1, u_2, t_3, t_6 and u_4, transition u_6 can fire unlimitedly. Net N enables behavior in net M that is not considered by the soundness check on M as the transitions u_4, u_5, u_6, u_7 and u_8 are dead.

Thus the simplest synchronizing net M as depicted in Fig. 2(c) is not a proper approximation of each sound workflow. For the AG-part of soundness, we conclude that net N should not contain transitions (like the ones we have just discovered) that may lead M into unexplored behavior.

3.4 Some Transitions Should Not Occur

The previous subsection suggests to check that N can only produce tokens on place p (using action p) and consume tokens from place p (using action p') in the orders described in Fig. 4. The initial state is s_0, and both the solid and the dashed transitions are permitted. For readability reasons, each state is annotated with the number of tokens in places p and q. Without loss of generality, we assume that each transition in N performs at most one action. In particular, in states s_4, s_5 and s_7, Fig. 4 excludes producing a token in q for the second time before any token has been consumed from p; thus excluding the example in Fig. 3(a).

However, this is not a sufficient condition. An example refinement $N \odot M$ is depicted in Fig. 3(b). In net N, transition t is crucial for termination; its firing implies that N has produced a token on p, but not yet on q. Net N only executes actions in the order specified in Fig. 4, but in $N \odot M$, net M eliminates the path to termination from net N. So M imposes additional synchronization on net N.

Thus the solid and dashed transitions in Fig. 4 are not a proper approximation of each sound workflow. For the EF-part of soundness, we conclude that net N should only use transitions that cannot be excluded by any M.

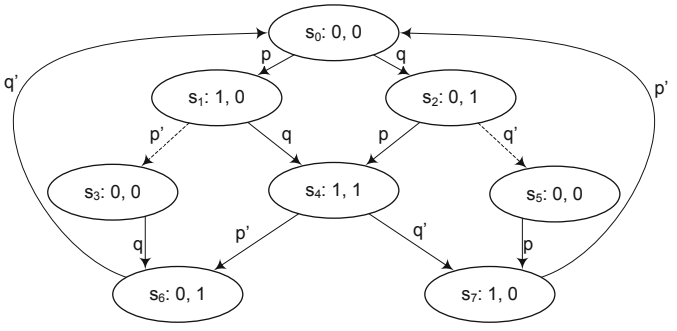


Fig. 4. May/exit transition system

3.5 Some Transitions Should Occur

The previous subsection suggests to check that also from every reachable non-final state of N, a final state can be reached using only the solid transitions in Fig. 4. In particular, in state s_1, this excludes that for termination of net N first a token from p must be consumed before any token on q has been produced.

Fig. 4 contains a may/exit transition system, where all transitions are may-transitions, and the solid transitions are also exit-transitions. Thus, we obtain a sufficient condition for synchronizable places p and q in N, viz., N should be sound under the restrictions from the may/exit transition system in Fig. 4. That is, for the AG-part of soundness both the exit- and the may-transitions can be used, whereas for the EF-part of soundness only the exit-transitions can be used.

In relation to modal (may/must) transition systems [15], the may-transitions correspond, whereas the exit- and must-transitions are unrelated. In relation to game theory [8], the may-transitions are "conserving", and the exit-transitions are "equalizing". A detailed study of may/exit transition systems is outside the scope of this paper.

4 Homogeneous Solution

The refinement described in Section 3 depends on a sound workflow for M and a special kind of soundness for N that takes into account the may/exit transition system from Fig. 4. In this section we show how these criteria can be formulated in a homogeneous way as two checks for (traditional) soundness. We start by defining place refinement in terms of composition of open nets.

4.1 Refinement in Terms of Composition

Suppose a net N is given with places p and q as in Fig. 5(a). To separate the incoming and outgoing arcs from these places we apply fusion of series places [23] and obtain Fig. 5(b). By definition of composition of asynchronous interfaces, this is equal to Fig. 5(c).

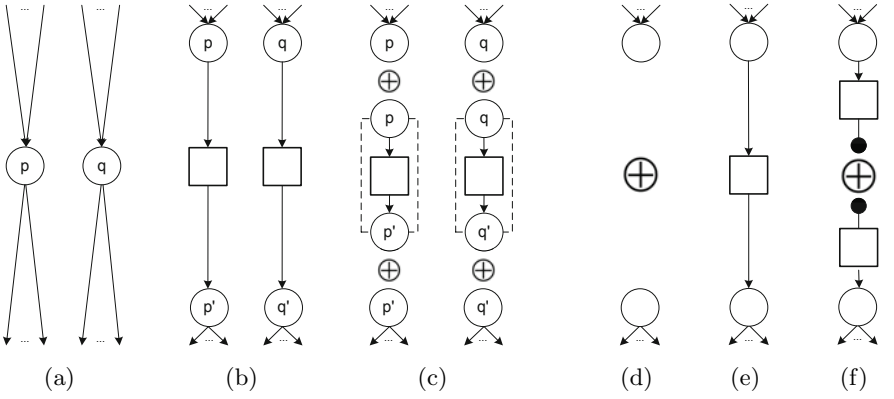


Fig. 5. Refinement in terms of composition

Thus each net N is equivalent to a similar open net N' composed with the open net from Fig. 2(b). The refinement of net N with an open net M (with an asynchronous interface), denoted by $N \odot M$, is defined as $N' \oplus M$.

Although asynchronous interfaces are most natural in Petri nets, our results are easier to explain in terms of synchronous interfaces. Therefore we show how two nets with an asynchronous interface can be translated into two nets with a synchronous interface, in such a way that the compositions are equivalent.

Consider any pair of corresponding interface places from the two nets as in Fig. 5(d). After composing them, we apply fusion of series places [23] and obtain Fig. 5(e). By definition of composition of synchronous interfaces, this is equal to Fig. 5(f). So every asynchronous interface place in the open nets N' and M becomes an internal place that is connected by a transition to a synchronous interface port (indicated by a black dot). Thus we transform open nets N' and M into open nets N'' and M'' such that $N \odot M$ is equivalent to $N'' \oplus M''$. In what follows, we use N and M to refer to N'' and M''.

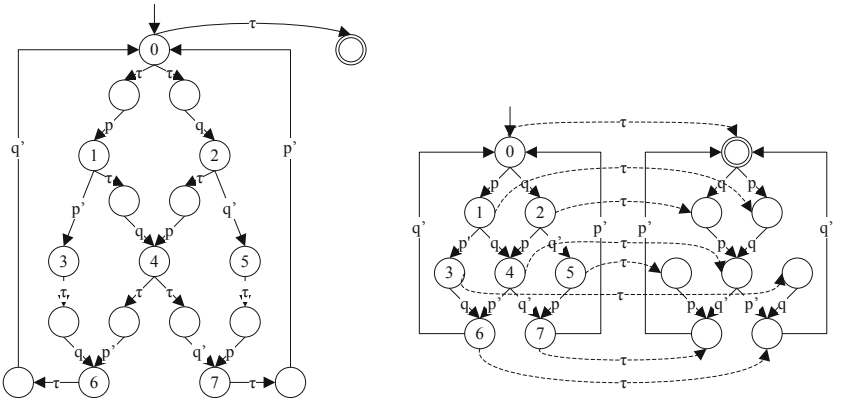
4.2 Two Checks for Soundness

In this section we propose a pair of open nets E (environment) and T (test) with the same synchronous interface as N and M. The idea is to conclude that $N \oplus M$ is sound if both $N \oplus T$ and $E \oplus M$ are sound:

$$(\forall N, M :: \ SD.(N \oplus T) \ \wedge \ SD.(E \oplus M) \ \ \Rightarrow \ \ SD.(N \oplus M))$$

For brevity reasons, in Fig. 6 we describe open nets E and T using an LTS (where double circles denote final states) with some additional transitions; such an LTS can be translated to an open net with a synchronous interface (where every transition has one incoming and one outgoing arc). For every numbered state i and action a mentioned in Fig. 6(c), a transition $i \xrightarrow{a} b$ should be added (called a "fact"-transition in [11]), where state b is a deadlock state.

In the remainder of this section we motivate these nets E and T using the insights from Section 3. In Section 5 we formally show their correctness, and the way in which net T can be computed from net E.



(a) Environment E (b) Test T

	0	1	2	3	4	5	6	7
E	p', q'	q'	p'	p', q'		p', q'	p'	q'
T		p	q	p	p, q	q	p, q	p, q

(c) Additional fact transitions

Fig. 6. State machines for the pair $\langle E, T \rangle$

Net E: The condition on M in Section 3 is that Fig. 2(a) yields a sound workflow. Definition 3 (Workflow) requires that all nodes are on a path from i to f, which, in combination with soundness, guarantees that after M has produced all output tokens, all non-output places of net M are empty. As the internal places of M cannot be accessed using composition, the overall structure of net E in Fig. 6(a) checks that M is sound even when executed multiple times. Note that this condition is slightly more liberal.

The τ-transitions in states 1 and 2 indicate that M does not need to be able to produce tokens in p' or q' before consuming a token from both p and q. The τ-transitions in states 0 and 4 indicate that M should not depend on the order in which tokens are produced in p and q, or consumed from p' and q'.

The fact transitions check a condition related to M modeling a pair of services, viz., whether M cannot produce a token in place p' too early (before consuming a token from place p); and similarly for q and q'.

Net T: The condition on N in Section 3 is that N is sound under the restrictions from the may/exit transition system in Fig. 4, and that all transitions that are connected to p and q occur in Fig. 4. Net T in Fig. 6(b) is motivated by the first part, and therefore consists of two sides: the left side contains both the may and the exit transitions, while the right side contains only exit transitions. The initial state is at the left, the final state is at the right, and there are only τ-edges from left to right. This motivates that net T checks that N is sound under the restrictions from Fig. 4.

What remains is to ensure that N does not contain transitions that do not occur in Fig. 4. The fact transitions in T check that the transitions p and q (that

produce a token in a place) do not occur as specified in Fig. 4. We do not need such a check for transitions p' and q' (that consume a token from a place), as in these cases the place is guaranteed to be empty.

5 Generalizing Theory

In this section we focus on the approach from Section 4. We give a theoretical foundation, and show a way in which valid pairs of nets can be constructed. In this way we generalize the notion of synchronizable pairs of places.

5.1 Foundation

Suppose two open nets N and M are given, and we want to prove that their composition is sound, denoted by $SD.(N \oplus M)$. We aim for a sufficient condition that can be split into parts referring only to N or M, but not both.

Let us calculate for any two open nets N and M:

$$
\begin{array}{ll}
& SD.(N \oplus M) \\
\equiv & \{\ \text{“}\Leftarrow\text{” Definition 6 (Accordance pre-order); “}\Rightarrow\text{” instantiation } E := N\ \} \\
& (\exists E ::\ N \leq E\ \wedge\ SD.(E \oplus M)) \\
\equiv & \{\ \text{Proposition 1 (Deciding accordance using maximal controller)}\ \} \\
& (\exists E ::\ SD.(N \oplus mc.E)\ \wedge\ SD.(E \oplus M))
\end{array}
$$

Note that conjunct $SD.(E \oplus M)$ guarantees that net E is controllable, and hence $mc.E$ is defined. For every open net E we thus obtain a sufficient condition for proving $SD.(N \oplus M)$, viz.,

$$(\forall E, M, N ::\ SD.(N \oplus mc.E)\ \wedge\ SD.(E \oplus M)\ \ \Rightarrow\ \ SD.(N \oplus M))$$

Alternatively, suppose the open nets E and M are given. We want to prove that $SD.(E \oplus M)$ is an exact condition for concluding that for each net N such that $SD.(N \oplus mc.E)$ holds, also $SD.(N \oplus M)$ holds.

Theorem 1.

$$
\begin{array}{l}
(\forall E, M ::\ SD.(E \oplus M)\ \equiv\ (\forall N ::\ SD.(N \oplus mc.E)\ \Rightarrow\ SD.(N \oplus M))) \\
(\forall E, N ::\ SD.(N \oplus mc.E)\ \equiv\ (\forall M ::\ SD.(E \oplus M)\ \Rightarrow\ SD.(N \oplus M)))
\end{array}
$$

Proof. We justify the two equivalences $\equiv$ by proving two implications:

$\Rightarrow$: follows from our introductory result (using quantifier logic);
$\Leftarrow$: follows from instantiations $N := E$ and $M := mc.E$ respectively. □

In fact, “$\Leftarrow$” uses that $mc.E$ is a controller, and “$\Rightarrow$” uses that it is maximal.

5.2 Computing Maximal Controllers

Maximal controllers are related to canonical duals [7] for which there is a trivial computation, but this does not apply in our setting. In [19] we have shown how to construct a maximal controller if the behavioral property is deadlock freedom.

This was based on operating guidelines [16] for deadlock freedom. As far as we know, there are no published results yet wrt. soundness for maximal controllers nor for operating guidelines. In what follows we compute a finite representation of a maximal controller for soundness, but only for a class of open nets.

The open nets that we consider are a generalization of net E in Fig. 6(a) or net T in Fig. 6(b). For describing the parameters, we focus on such a net E. The states contain a set I of core states (which are numbered in Fig. 6(a)), including the initial state b. There is a subset F of I that contains the core states with a τ transition to a final state without outgoing transitions.

The transitions from each core state $i: \ i \in I$ can be characterized by three sets: set $E.i$ contains the non-τ actions from state i, set $C.i$ contains the non-τ actions from state i for which there is a dedicated τ transition from i, and set $D.i$ contains the fact non-τ actions from state i that lead to a deadlock. Finally, $W.i.a$ indicates the next core state after doing action a in core state i;

Definition 8 (Pair of nets $\langle X, Y \rangle$). *Given a set I of core states, an initial core state $b: \ b \in I$, and a set $F: \ F \subseteq I$ of final core states. Furthermore, for every $i: \ i \in I$, three sets $E.i$, $C.i$ and $D.i$ of actions are given, and a successor function $W.i.a$ with result type I for $a: \ a \in E.i \cup C.i$.*

A pair of nets $\langle X, Y \rangle$ consists of two open nets X and Y. Open net X has the following LTS semantics (for any $\delta: \ \delta \notin I$ and $\omega: \ \omega \notin I$):

$$
\begin{array}{lcl}
S & = & \{X.i \mid i \in I\} \ \cup \ \{X.i.a \mid i \in I \ \wedge \ a \in C.i\} \ \cup \ \{X.\delta, X.\omega\} \\
\rightarrow & = & \{(X.i, a, X.(W.i.a)) \mid a \in E.i\} \\
 & & \quad \cup \ \{(X.i, \tau, X.i.a) \mid a \in C.i\} \ \cup \ \{(X.i.a, a, X.(W.i.a)) \mid a \in C.i\} \\
 & & \quad \cup \ \{(X.i, a, X.\delta) \mid a \in D.i\} \ \cup \ \{(X.i, \tau, X.\omega) \mid i \in F\} \\
\Omega & = & \{X.\omega\} \qquad s_i \ = \ X.b
\end{array}
$$

Open net Y has the following LTS semantics (for any $\delta: \ \delta \notin I$):

$$
\begin{array}{lcl}
S & = & \{Y.i \mid i \in I\} \ \cup \ \{Z.i \mid i \in I\} \ \cup \ \{Y.\delta\} \\
\rightarrow & = & \{(Y.i, a, Y.(W.i.a)) \mid a \in E.i \cup C.i\} \ \cup \ \{(Z.i, a, Z.(W.i.a)) \mid a \in C.i\} \\
 & & \quad \cup \ \{(Y.i, a, Y.\delta) \mid a \in \overline{E.i \cup C.i \cup D.i}\} \ \cup \ \{(Y.i, \tau, Z.i) \mid i \in I\} \\
\Omega & = & \{Z.i \mid i \in F\} \qquad s_i \ = \ Y.b
\end{array}
$$

This definition provides a procedure for computing Y if X is given, or X if Y is given. Net E in Fig. 6(a) and net T in Fig. 6(b) form a pair of nets $\langle E, T \rangle$ as described in Definition 8.

In the remainder of this section we provide a condition on pairs of nets $\langle X, Y \rangle$ that guarantees that nets X and Y are a maximal controller of each other. To this end we need to prove that net Y is a controller of net X, and net Y is larger than each controller of X; and vice versa.

The first requirement boils down to checking soundness of $X \oplus Y$. We focus on the last requirement, which can be formalized as

- $(\forall M :: \ SD.(X \oplus M) \ \Rightarrow \ M \leq Y)$, and
- $(\forall N :: \ SD.(N \oplus Y) \ \Rightarrow \ N \leq X)$,

which are equivalent to the following symmetric formalization:

$$(\forall M, N :: \ SD.(N \oplus Y) \ \wedge \ SD.(X \oplus M) \ \Rightarrow \ SD.(N \oplus M))$$

Before proving this in Lemma 2, we first prove a supporting lemma. For a path σ, we use $\pi_N.\sigma$ to denote the projection of σ on the steps by N; we use $s \xrightarrow{\sigma} t$ to denote that there is a path σ from state s to state t. In the context of synchronous interfaces, each state that is reachable in the composition $N \oplus M$ can be described as a pair (s_N, s_M) consisting of a state s_N in N and a state s_M in M.

Lemma 1. *Given any pair of nets $\langle X, Y \rangle$ as in Definition 8, and any two open nets M and N, such that $SD.(N \oplus Y)$ and $SD.(X \oplus M)$. For any path σ and states s_N and s_M such that $N \oplus M \xrightarrow{\sigma} (s_N, s_M)$, there exists a core state i and paths σ_1 and σ_2 such that:*

$$N \oplus Y \xrightarrow{\sigma_1} (s_N, Y.i) \wedge X \oplus M \xrightarrow{\sigma_2} (X.i, s_M) \wedge \pi_N.\sigma = \pi_N.\sigma_1 \wedge \pi_M.\sigma_2 = \pi_M.\sigma$$

Proof. We prove this using structural induction on σ. In the basis $\sigma = \epsilon$ we choose $\sigma_1 = \sigma_2 = \epsilon$ and $i = b$. Each appended internal step of either N or M in σ can be appended to σ_1 or σ_2 respectively without affecting i.

Each appended synchronized step a in σ is a step of both M and N, and hence it should be appended to both σ_1 and σ_2. As $SD.(N \oplus Y)$ and $SD.(X \oplus M)$, a is not a fact transition in state $X.i$ nor in state $Y.i$, i.e., a is not in $D.i$ nor in $\overline{E.i \cup C.i \cup D.i}$; hence a is in $E.i$ or $C.i$. Both $X.i$ and $Y.i$ can perform this step (after inserting a τ-step in σ_2 in case $a \in C.i$), and resulting in state $X.(W.i.a)$ and $Y.(W.i.a)$ respectively. □

Lemma 2. *For every pair of nets $\langle X, Y \rangle$ as in Definition 8:*

$$(\forall M, N :: \ SD.(N \oplus Y) \ \wedge \ SD.(X \oplus M) \ \Rightarrow \ SD.(N \oplus M))$$

Proof. Assume $SD.(N \oplus Y)$ and $SD.(X \oplus M)$, and let us focus on $SD.(N \oplus M)$. Soundness denotes that after every path there is a path to a final state. From Lemma 1 we can conclude that every path in $N \oplus M$ to any state (s_N, s_M) can be mimicked using paths to $(s_N, Y.i)$ and $(X.i, s_M)$ in $N \oplus Y$ and $X \oplus M$.

What remains is to construct a path from state (s_N, s_M) to a final state. At this point we use the τ-step from $Y.i$ to $Z.i$. As $N \oplus Y$ is sound, there is a path from $(s_N, Z.i)$ to a final state $(s'_N, Z.i')$, with $i' \in F$; in Y such a path can only use C-actions.

As $X \oplus M$ is sound (and hence deadlock free), we can use the τ-steps to the C-actions in X to construct a path in $X \oplus M$ to a state $(X.i', s'_M)$ using the same synchronized steps as the path in $N \oplus Y$. From state $(X.i', s'_M)$ the state $(X.\omega, s'_M)$ can be reached using a τ-step. As $X \oplus M$ is sound, this means that from state s'_M in M there is a path to a final state s''_M using only internal steps. Using a proper synchronization of these paths, we obtain a path in $M \oplus N$ to a final state (s'_N, s''_M). □

Theorem 2. *Given a pair of nets $\langle X, Y \rangle$ as in Definition 8. If the composition $X \oplus Y$ is sound, then nets X and Y are a maximal controller of each other.*

The composition $T \oplus E$ for the pair $\langle T, E \rangle$ in Fig. 6 is sound. Given the disjointness of C, D, E and $\overline{E.i \cup C.i \cup D.i}$, and as the C's are non-empty, there are no reachable deadlocks. From every reachable state there is a path to a final state that only uses C transitions. Hence T and E are maximal controllers.

Using the construction from this section, we can generalize Section 4 from pairs of synchronizable places to larger numbers of synchronizable places. In these cases, the nets E and T become bigger; net E is probably the simplest one to modify manually, whereas net T can better be computed using Definition 8.

6 Conclusions and Further Work

In the context of service compositions, we have developed conditions on a refined net N and a refining net M in isolation such that the refinement of N by M is sound. We have generalized these techniques to a larger class of refinements, and proved their correctness in terms of composition and maximal controllers.

It is further work to combine our techniques (aimed at vertical modularization) with the techniques from [2] for horizontal modularization. Furthermore, we want to investigate which other temporal properties, besides soundness, can be preserved by (extensions of) our method.

Our techniques are based on pairs of nets that are each others maximal controller. These nets can be considered as tests, and thus have a much wider application area. For example, a sound pair $\langle X, Y \rangle$ can be seen as a contract between two organizations. If one organization develops a service M that is sound with X, and the other independently develops a service N that is sound with Y, then the composition of M and N is also guaranteed to be sound. In this way, X and Y can be seen as test stubs and skeletons for M and N.

References

1. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. van der Aalst, W.M.P., van Hee, K.M., Massuthe, P., Sidorova, N., van der Werf, J.M.E.M.: Compositional service trees. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 283–302. Springer, Heidelberg (2009)
3. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. The Computer Journal (2009)
4. Brauer, W., Gold, R., Vogler, W.: A survey of behaviour and equivalence preserving refinements of Petri nets. In: ATPN 1990. LNCS, vol. 483, pp. 1–46 (1991)
5. Bravetti, M., Tennenholtz, M.: Contract based multi-party service composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)
6. Brinksma, E.: A theory for the derivation of tests. In: Protocol Specification, Testing, and Verification VIII, pp. 63–74. North-Holland, Amsterdam (1988)
7. Castagna, G., Dezani-Ciancaglini, M., Giachino, E., Padovani, L.: Foundations of session types. In: PPDP 2009, pp. 219–230. ACM, New York (2009)

8. Chow, Y.S., Robbins, H., Siegmund, D.: The Theory of Optimal Stopping: Great Expectations. Houghton Mifflin Company (1971)
9. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs 1981. LNCS, vol. 131, pp. 52–71 (1982)
10. Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. MIT Press, Cambridge (1989)
11. Genrich, H.J., Thieler-Mevissen, G.: The calculus of facts. In: MFCS 1976. LNCS, vol. 45, pp. 588–595 (1976)
12. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and separability of workflow nets in the stepwise refinement approach. In: van der Aalst, W.M.P., Best, E. (eds.) ATPN 2003. LNCS, vol. 2679, pp. 337–356. Springer, Heidelberg (2003)
13. Kindler, E.: A compositional partial order semantics for Petri net components. In: ATPN 1997. LNCS, vol. 1248, pp. 235–252 (1997)
14. Kindler, E., Martens, A., Reisig, W.: Inter-operability of workflow applications: Local criteria for global soundness. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) BPM 2000. LNCS, vol. 1806, pp. 235–253. Springer, Heidelberg (2000)
15. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS 1988, pp. 203–210 (1988)
16. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: Kleijn, J., Yakovlev, A. (eds.) ATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
17. Malik, R., Streader, D., Reeves, S.: Conflicts and fair testing. Journal of Foundations of Computer Science 17(4), 797–813 (2006)
18. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics 1(3), 35–43 (2005)
19. Mooij, A.J., Parnjai, J., Stahl, C., Voorhoeve, M.: Constructing substitutable services using operating guidelines and maximal controllers (2010) (accepted for WS-FM 2010)
20. Mooij, A.J., Stahl, C., Voorhoeve, M.: Relating fair testing and accordance for service replaceability. Journal of Logic and Algebraic Programming 79(3–5), 233–244 (2010)
21. Mooij, A.J., Voorhoeve, M.: Proof techniques for adapter generation. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 207–223. Springer, Heidelberg (2009)
22. Mooij, A.J., Voorhoeve, M.: Trading off concurrency to generate behavioral adapters. In: ACSD 2009, pp. 109–118. IEEE, Los Alamitos (2009)
23. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
24. Rensink, A., Vogler, W.: Fair testing. Information and Computation 205(2), 125–198 (2007)
25. Schmidt, K.: LoLA: A low level analyser. In: Nielsen, M., Simpson, D. (eds.) ATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000)
26. Siegeris, J., Zimmermann, A.: Workflow model compositions preserving relaxed soundness. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 177–192. Springer, Heidelberg (2006)
27. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. 68(9), 819–833 (2009)
28. Suzuki, I., Murata, T.: A method for stepwise refinement and abstraction of Petri nets. Journal of Computer and System Sciences 27, 51–76 (1983)
29. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)

Formal Semantics and Implementation of BPMN 2.0 Inclusive Gateways

David Raymond Christiansen, Marco Carbone, and Thomas Hildebrandt*

IT University of Copenhagen,
Rued Langgaards Vej 7,
2300 Copenhagen, Denmark

Abstract. We present the first direct formalization of the semantics of inclusive gateways as described in the Business Process Modeling Notation (BPMN) 2.0 Beta 1 specification. The formal semantics is given for a minimal subset of BPMN 2.0 containing just the inclusive and exclusive gateways and the start and stop events. By focusing on this subset we achieve a simple graph model that highlights the particular non-local features of the inclusive gateway semantics. We sketch two ways of implementing the semantics using algorithms based on incrementally updated data structures and also discuss distributed communication-based implementations of the two algorithms.

1 Introduction

Business Process Modeling Notation (BPMN), a standardized notation for representing processes within organizations, is soon to be released in a major new revision. According to the draft BPMN 2.0 specification,

> The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. Thus, BPMN creates a standardized bridge for the gap between the business process design and process implementation [5, p. 1].

Because BPMN seeks to serve as a kind of universal communication tool for business processes, it is vitally important that all parties agree on the meaning of a BPMN diagram. The BPMN 2.0 specification provides a rather detailed, but still only informal description of its semantics [5, p. 389].

The so-called *inclusive gateways* of BPMN seem particularly challenging to provide semantics for, since a non-trivial (and non-local) backwards search in

* This research is supported by the Danish Research Agency through the FIRST Graduate School (www.first.dk), the Jingling Genie project (http://sites.google.com/site/jinglinggenie/) and the Computer Supported Mobile Adaptive Business Processes project (grant #274-06-0415, www.cosmobiz.dk).

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 146–160, 2011.

the flow graph is included in the specification of their semantics. This is similar to the OR-joins of YAWL and EPC which have been the subject of several papers aiming to clarify their non-local semantics [6,1,3]. In this work, we focus on a small subset of BPMN 2.0, called BPMN_{inc} (BPMN inclusive) which just includes the primitives needed to illustrate the complexity of inclusive gateways and formalize their semantics in a way that can be generalized to full BPMN 2.0.

BPMN_{inc} is defined in Section 2. A formal semantics is provided for BPMN_{inc} in Section 3, followed by a discussion of how it can be implemented efficiently and in a distributed manner. Related work is discussed in Section 4.

2 BPMN_{inc} and Its Informal Semantics

BPMN process diagrams contain a large number of different graphical elements. Broadly speaking, there are four classes of elements that are of interest when designing semantics:

Sequence Flow describes the order in which various parts of the process occur.
Events represent things that can happen during a process, such as a message being sent or a timer.
Activities represent work performed by a company, and can either be atomic (in which case they are called *tasks*) or they can represent another process diagram.
Gateways provide flow control within a process diagram [5, p. 21].

With the exception of Sequence Flow, there are multiple variations of each of the above elements, providing for different kinds of flow control and allowing representation of different kinds of business activities.

For purposes of this paper, only a small subset of BPMN that is sufficient to illustrate certain difficult properties will be used. BPMN_{inc}, the subset, contains:

- Sequence flow
- Exclusive gateways
- Inclusive gateways
- Start events
- End events

Above, a gateway is exclusive when it behaves as an exclusive conditional while it is inclusive when its activation depends on further conditions on the incoming flows as well as allowing for multiple parallel outcomes. Start and end events model initiation and termination of BPMN processes. In the following, certain aspects of BPMN_{inc} will not be defined in full detail. For example, the conditions that determine which outgoing sequence flow should be chosen after a gateway is activated are simply assumed to exist and be subject to evaluation, giving either a true or false result, while the mechanism of this evaluation remains unspecified.

Activities are not included in BPMN_{inc}, as their possible effects are not modeled. For our purposes, an activity will be equivalent to an exclusive gateway with a single incoming and a single (default) outgoing sequence flow.

Finally, note that the parallel split from BPMN where the flow of execution is split into two parallel flows can just be seen as a special case of inclusive gateways, where all outgoing conditions evaluate to true and the default flow connects to the end event. Parallel join is not straightforwardly encodable using inclusive gateways, but it is however straightforward to include in our semantics.

2.1 Sequence Flow and Tokens

Sequence flow represents the order in which the execution of a BPMN process occurs. [5, p. 21] It is represented as an arrow.

We follow the BPMN 2.0 specification in representing the execution state of a process with tokens on sequence flow[5], which is represented graphically as a small solid black circle placed next to the sequence flow. When the sequence flow before some element of a process diagram receives a token, then the element is activated in some way dependent on the precise type of element. The control-flow semantics of the gates are then given by the tokens required on their incoming sequence flow and the tokens produced on the outgoing sequence flow. Note that a sequence flow may have more than one token.

2.2 Exclusive Gateways

The semantics of exclusive gateways are quite uncomplicated. When a token arrives on any incoming sequence flow, it evaluates the conditions on the outgoing sequence flow until it finds one that returns true. It then places a token on that sequence flow and stops evaluating conditions. If no condition evaluates to true, then the sequence flow marked as default receives the token. [5, p. 401]

In BPMN_{inc}, every exclusive gateway must have a default outgoing sequence flow, which is indicated by placing a slash through the line immediately next to the gateway. In the exclusive gateway in Fig. 1, when a token arrives on *any one*

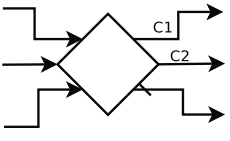


Fig. 1. An exclusive gateway

of the incoming sequence flow on the left, C_1 is evaluated. If it is true, a token is emitted on the sequence flow that C_1 is associated with. If it is not true, then C_2 is evaluated, and if it is true, then a token is emitted on the sequence flow attached to C_2. Finally, if no condition evaluates to true, then a token is emitted on the default sequence flow.

2.3 Inclusive Gateways

The BPMN inclusive gateway is problematic because it requires a search to be made for tokens upstream of it.[1]. To quote the specification:

> The Inclusive Gateway is activated if
> – At least one incoming sequence flow has at least one *Token* and

[1] Note that other gateways, such as the complex gateway, share this behavior.

– for each empty incoming sequence flow, there is no *Token* in the graph anywhere upstream of this sequence flow, i.e., there is no directed path (formed by Sequence Flow) from a Token to this sequence flow unless
  - the path visits the inclusive gateway or
  - the path visits a node that has a directed path to a non-empty incoming sequence flow of the inclusive gateway. [5, p. 401]

BPMN_{inc} includes all this behavior. Note that the specification is independent of which other events, activities or gateways are allowed in the diagram. Consequently, our formal semantics for inclusive gateways can be straightforwardly extended to any superset.

According to the above definition, in the process shown in Fig. 2, if there is a token on edges A and B, then the rightmost inclusive gateway is allowed to activate because there is a directed path from the topmost exclusive gateway to A. However, if the topmost exclusive gateway fires first and sends a token on its top sequence flow, then the gateway labeled α must complete, depositing a token at C, before the rightmost inclusive gateway can fire. The example thus also illustrates that the BPMN 2.0 specification of the behavior of inclusive gateways may lead to race conditions.

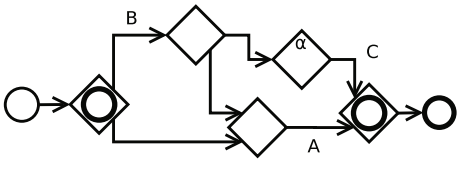


Fig. 2. Process with inclusive gateways

The other exception in the search defined above relates to gateways in cycles. Fig. 3 shows one such process. The inclusive gateway can fire if there is a token on the first sequence flow (immediately after the start event) even though the top incoming sequence flow does not have a tokens.

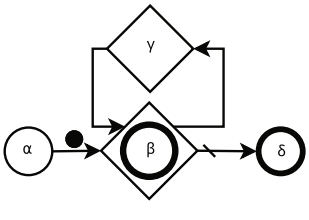


Fig. 3. A simple process with a loop

A simple approach to a token-based semantics, in which gates determine whether to fire based only upon which of the incoming sequence flow contain

tokens, is clearly unable to implement the requirements of the specification without resorting to some kind of global information about the flow of control in the rest of the process.

After the inclusive gateway fires, the conditions on all outgoing sequence flow are evaluated. If any of them are true, then *all* sequence flow whose conditions are true receive tokens. If none of them are true, then the default outgoing sequence flow receives a token.

2.4 Start and End Events

A start event is responsible for emitting the first token that starts the process. BPMN has relatively complex semantics for running multiple parallel instances of subprocesses. Because BPMN_{inc} has no concept of subprocesses, start events need only emit the first token.

Likewise, no specific semantics for the end of a subprocess are necessary in BPMN_{inc}. Therefore, end events simply consume tokens, and the process is complete when there are no tokens remaining.

3 Formal Semantics for BPMN_{inc}

In this section, we define a formal semantics for BPMN_{inc}. Unlike the other possible approaches to providing a formal semantics for BPMN (see Section 4), the semantics provided here does not attempt to translate the BPMN_{inc} process to another formal system. Instead, semantics is given operationally directly as a token-based semantics for the BPMN_{inc} process graph.

Throughout this section, the process in Fig. 3 will be used to demonstrate the formalization. The letters α, β, γ and δ are not a part of BPMN_{inc}, but they provide a means of referring to individual gateways.

3.1 BPMN_{inc} Process Graphs

Below, we give the formal definition of BPMN_{inc} process graphs:

Definition 1 (BPMN_{inc} **Process).** [2] *A process* $\mathcal{P}$ *is a tuple* $(N, \mathcal{L}, S, \preceq, C, M)$ *such that:*

- N *is a set of nodes;*
- *the labeling* $\mathcal{L} : N \to \{Excl, Incl, Start, End\}$ *maps nodes in* N *to either Excl (BPMN exclusive gateways), Incl (BPMN inclusive gateways), Start (start events), and End (end events);*
- $S \subseteq N \times N$ *is a set of sequence flows such that* $(n, n') \in S$ *implies* $\mathcal{L}(n) \neq End$ *and* $\mathcal{L}(n') \neq Start$*;*

[2] This definition is somewhat analogous to that of Petri nets[7], where N corresponds to transitions, and S corresponds to places, except that the sequence flow connects exactly one node to another node.

- $\preceq \subseteq S \times S$ *is an ordering relation over* S *such that*
  - *for all* n *if* $(n, n') \in S$ *and* $(n, n'') \in S$ *then either* $(n, n') \preceq (n, n'')$ *or* $(n, n'') \preceq (n, n')$ *(*$\preceq$ *is total over the outgoing sequence flows for any given gateway);*
  - *and* $(n_1, n_2) \preceq (n_3, n_4)$ *implies* $n_1 = n_3$ *(*$\preceq$ *does not relate sequence flows with different source gateways);*
- $C : S \to Cond$ *is a partial map (defined on the outgoing sequence flows of inclusive and exclusive gateways) to (an assumed set of) conditions such that the maximal (w.r.t. the ordering* $\preceq$*) outgoing flow for each gateway always has the condition true; and*
- $M : S \to \mathbb{N}$ *is a marking of* S *with tokens.*

Above, $\preceq$ is an ordering relation defined such that the outgoing sequence flow of any node is totally ordered. The maximal outgoing flow of exclusive and inclusive gateways w.r.t. this ordering plays the special role as the *default* flow. Graphically, this is represented by the layout order of the outgoing sequence flow, with a dash through the line of the default flow (if there are more than one outgoing flow). In the rest of the paper we will let $D \triangleq \{s \mid \mathcal{L}(fst(s)) \in \{Excl, Incl\} \wedge \forall s' : s \preceq s' \text{ implies } s' = s\}$ be the set of all default flows. We can then use this ordering when implementing the exclusive gateway in order to determine the order in which the conditions on the outgoing sequence flow should be evaluated. C represents a mapping from sequence flow to logical conditions that are evaluated in order to determine which outgoing sequence flow of some gateway receives tokens when that gateway fires. For our purposes, it suffices simply to define some condition $C(s)$ for some sequence flow $s \in S$ to be true just in case some evaluation function $eval(C(s))$ returns true. Note that it is not strictly necessary to introduce C and $eval$, as the rules could simply be rewritten to be nondeterministic, yielding a simpler semantics with equivalent behavior. However, the current formulation is closer to the semantics of full BPMN, and it makes clear exactly how this particular nondeterminism is to be removed when expanding the subset.

Example 1. *The process* $\mathcal{P}_1$ *in Fig. 3 is represented as the tuple* $(N, \mathcal{L}, S, \preceq, C, M)$*, where*

$$
\begin{aligned}
N &= \{\alpha, \beta, \gamma, \delta, \} \\
\mathcal{L} &= \{(\alpha, Start), (\beta, Incl), (\gamma, Excl), (\delta, End)\} \\
S &= \{(\alpha, \beta), (\beta, \gamma), (\beta, \delta), (\gamma, \beta)\} \\
C &= \{((\beta, \gamma), C_{\beta,\gamma}), \} \\
M &= \{((\alpha, \beta), 1), ((\beta, \gamma), 0), ((\beta, \delta), 0), ((\gamma, \beta), 0)\} \\
&\quad \textit{and } (\beta, \gamma) \preceq (\beta, \delta) \textit{ is the only non-trivial pair in } \preceq
\end{aligned}
$$

where $C_{\beta,\gamma}$ *denotes some condition with associated evaluation function eval that returns either true or false. The set* D *of default flows given as the maximal outgoing flows w.r.t. the order* $\preceq$ *is* $\{(\beta, \delta), (\gamma, \beta)\}$.

Before giving the formal semantics of processes, we formalize the notions of incoming and outgoing sequence flows.

Definition 2 (Incoming/Outgoing Sequence Flow). *The incoming sequence flow of some node n, written $S_{in}(n)$, is defined as $S_{in}(n) = \{(n_0, n_1) \in S \mid n_1 = n\}$. The outgoing sequence flow S_{out} is defined as $S_{out}(n) = \{(n_0, n_1) \in S \mid n_0 = n\}$.*

Additionally:

Definition 3 (Source/Target Nodes on Sequence Flow). *The source and target nodes of some sequence flow $s = (n_0, n_1)$ are defined as $fst(s) = n_0$ and $snd(s) = n_1$.*

Example 2. *For $\mathcal{P}$ from Example 1 $S_{in}(\beta) = \{(\alpha, \beta), (\gamma, \beta)\}$ and $S_{out}(\beta) = \{(\beta, \gamma), (\beta, \delta)\}$.*

3.2 BPMN$_{\mathbf{inc}}$ Formal Semantics

The approach to semantics discussed in this section successfully implements the rules governing inclusive gateways informally discussed above. Evaluation is divided into two phases. The first phase of the evaluation consists of annotating each sequence flow with the set of paths from that sequence flow to each upstream token. In the second phase we use that information to determine if inclusive gateways can fire. Once a gateway fires, a token (and at most one) is obviously consumed.

The annotation map G has type $S \to 2^{N^*}$, or in other words, it maps sequence flow to sets of sequences of nodes. To avoid confusion, sequences of nodes are written in square brackets, where [] represents the empty sequence. G is computed with algorithm 3.1. One way to picture the operation of the algorithm (which we will elaborate a bit more in Section 3.3 below) is by imagining that each token sends a message down its sequence flow. That message, which starts with an empty payload, accumulates the unique identifiers of each of the gateways that it crosses. The message travels out of *all* of the outgoing sequence flow of each gateway it enters. If the gateway that the message is about to cross is already listed in the message payload, then it stops.

The algorithm consists of two working procedures AddToPath and StepPathAnnotation along with the main code contained in **main**. Procedure AddToPath simply takes a path of nodes $[a_0, a_1, \ldots, a_n]$ and appends a new node b at its end only if b is not already present. The other procedure, namely StepPathAnnotation takes a flow annotation G as an argument and returns a new one by updating G according to the marking M. More precisely, each sequence flow s containing a token (that is, where $M(s) > 0$) is annotated with the empty string. On the other hand, each edge with no tokens is updated according to the annotations of upstream sequence flows: if (n_0, n) and (n, n_1) are in S then $G((n, n_1))$ is also updated with AddToPath$(G((n_0, n)), n)$. Finally, the code in **main** computes the least fixed point of StepPathAnnotation.

Algorithm 3.1. ComputePathAnnotations(S, N, M)

procedure AddToPath$([a_0, a_1, \ldots, a_n], b)$
/*Add b to the end of the path only if b is not in the path already*/
if $b \in \{a_0, a_1, \ldots, a_n\}$
then $p \leftarrow [a_0, a_1, \ldots, a_n]$
else $p \leftarrow [a_0, a_1, \ldots, a_n, b]$
return (p)

procedure StepPathAnnotation(G)
$G' \leftarrow \emptyset$ /*G' represents the new annotation to be generated from G. */
for each $s = (n_0, n_1) \in S$

do
- **if** $M(s) > 0$
  - **then**
    - $G'(s) \leftarrow \{[]\}$
    - /* Sequence flow containing a token always has an empty path to a token */
  - **else**
    - $p_s \leftarrow \emptyset$
    - **for each** $s' \in S_{in}(n_0)$ **do** $p_s \leftarrow p_s \cup \{\text{AddToPath}(p, n_0) \mid p \in G(s')\}$
    - $G'(s) \leftarrow p_s$
    - /* Each sequence flow gets incoming sequence flows' + the node on the left */

return (G')

main
$G_0 \leftarrow$ StepPathAnnotation$(\emptyset)$
repeat

do
- /* Iterate until the fixed point is found */
- $G_1 \leftarrow G_0$
- $G_0 \leftarrow$ StepPathAnnotation(G_0)

until $G_0 = G_1$
return (G_1)

Example 3. *Applying Algorithm 3.1 to $\mathcal{P}$ from Example 1 yields the map*

$$
\begin{aligned}
(\alpha, \beta) &\mapsto \{[]\} \\
(\beta, \gamma) &\mapsto \{[\beta], [\beta\gamma]\} \\
(\gamma, \beta) &\mapsto \{[\beta\gamma]\} \\
(\beta, \delta) &\mapsto \{[\beta], [\beta\gamma]\}
\end{aligned}
$$

This annotation is illustrated visually in Fig. 4

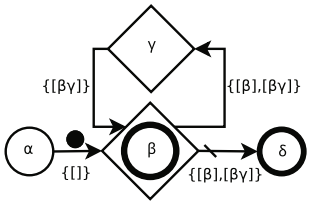


Fig. 4. Example path annotation

We can now state the following:

Theorem 1. *Algorithm 3.1 always terminates.*

Proof (Sketch). The fix point always exists simply because the topology of the process is finite, paths can at most contain each node once, and the function AddToPath is monotone. □

Given a BPMN_{inc} process $(N, \mathcal{L}, S, \preceq, C, M)$ with path annotation G, a new marking M' can be obtained by applying any of the following rules to any node $n = (a, t)$.

First, to help keep the definition of the rule covering inclusive gateways readable, the function *NotBlocking*, which determines whether a particular empty incoming sequence flow prevents an inclusive gateway from activating, is defined as follows:

Definition 4. *The predicate NotBlocking*(s) *is true if and only if*

$$\left(\begin{array}{ll} G(s) = \emptyset & \vee \\ \forall p \in G(s) : \left(\begin{array}{ll} snd(s) \in p & \vee \\ \exists s' \in S_{in}(snd(s)) : \left(\begin{array}{ll} M(s') > 0 & \wedge \\ \exists n \in p : Reachable(n, s') \end{array}\right) \end{array}\right) \end{array}\right)$$

where

$$Reachable(n, s) \quad \triangleq \quad \Big(n = fst(s) \quad \vee \quad \exists s' \in S_{in}(fst(s)) : Reachable(n, s')\Big)$$

NotBlocking corresponds to the conditions defined in the specification under which an inclusive gateway is allowed to fire even though it has empty incoming sequence flow. When listing the requirements for empty incoming sequence flow to inclusive gateways, the specification states that the gateway may only fire if there is no token upstream of the empty sequence flow, unless the path to the token visits the inclusive gateway or any node on the path is upstream of one of the incoming sequence flows with a token [5, p. 401]. In the definition of *NotBlocking*:

– The clause $G(s) = \emptyset$ corresponds to there being no token upstream of the sequence flow, as it means there are no paths from a token to s.

– The first disjunct within the universally quantified clause represents the situation in which the path in question visits the inclusive gateway.
– The second disjunct represents the path in question being upstream of an incoming sequence flow that has a token. This is because if a node that is upstream of the empty incoming sequence flow includes another node that is part of a path to a sequence flow with a token, then that node is upstream of that sequence flow.

We now give the formal semantics of BPMN_{inc} through three rules. In the following, we define $\delta_\phi = 1$ whenever the predicate ϕ is true and $\delta_\phi = 0$ otherwise. First, we give the rules for the activation of inclusive gateways:

$$(\textsc{Incl})\ \frac{\begin{array}{ll}\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s)) & \exists s_t \in S_{in}(n) : M(s_t) > 0 \\ \exists s_{ok} \in S_{out}(n) : (s_{ok} \notin D \wedge \mathit{eval}(C(s_{ok}))) & \mathcal{L}(n) = \mathit{Incl}\end{array}}{\begin{array}{c}\forall s \in S_{in}(n) :\ M'(s) = M(s) - \delta_{M(s)>0} \\ \forall s' \in S_{out}(n) :\ M'(s') = M(s') + \delta_{(s' \notin D \wedge \mathit{eval}(C(s')))}\end{array}}$$

$$(\textsc{Incl}_{\text{DF}})\ \frac{\begin{array}{ll}\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s)) & \exists s_t \in S_{in}(n) : M(s_t) > 0 \\ \neg\,(\exists s_{ok} \in S_{out}(n) : (s_{ok} \notin D \wedge \mathit{eval}(C(s_{ok})))) & \mathcal{L}(n) = \mathit{Incl}\end{array}}{\begin{array}{c}\forall s \in S_{in}(n) : M'(s) = M(s) - \delta_{M(s)>0} \\ \forall d \in S_{out}(n) : M'(d) = M(d) + \delta_{d \in D}\end{array}}$$

These rules construct a new marking M', which gives the marking of the ingoing and outgoing flows after the gateway has fired. ($\textsc{Incl}_{\text{DF}}$) activates the default outgoing sequence flow if no outgoing sequence flow with conditions receive tokens.

The clause $\mathcal{L}(n) = \mathit{Incl}$ in the premise of (Incl) simply restricts the rule to only apply to inclusive gateways. The condition $\exists s_t \in S_{in}(n) : M(s_t) > 0$ corresponds to the requirement in the specification that "At least one incoming sequence flow has at least one Token" [5, p. 401]. The condition $\forall s \in S_{in}(n) : (M(s) > 0 \vee \mathit{NotBlocking}(s))$ guarantees that each empty incoming sequence flow meets the requirements, as discussed above. Finally, the condition $(\exists s_{ok} \in S_{out}(n) : (S_{ok} \notin D \wedge \mathit{eval}(C(s_{ok}))))$ guarantees that the outgoing sequence flow should not be activated. In that case, ($\textsc{Incl}_{\text{DF}}$) should be activated instead. The two components of the conclusion of (Incl) are responsible for removing tokens from the incoming sequence flow and distributing them over the outgoing sequence flow according to the conditions, respectively.

The structure of ($\textsc{Incl}_{\text{DF}}$) is similar to that of (Incl). It simply states that if an inclusive gateway fires and none of its non-default outgoing sequence flow will receive a token, then the default sequence flow receives the token. Otherwise, it works as (Incl).

Exclusive gateways are much simpler:

$$\text{(Excl)}\ \frac{\begin{array}{ll} s \in S_{in}(n) & M(s) > 0 \\ s' \in S_{out}(n) & \mathit{eval}\,(C\,(s')) \\ \forall s'' \in S_{out}(n) : \mathit{eval}(C(s'')) \implies s' \preceq s'' & \mathcal{L}(n) = \mathit{Excl} \end{array}}{\begin{array}{c} \forall s_{in} \in S_{in}(n) : M'(s) = M(s) - \delta_{s_{in}=s} \\ \forall s_{out} \in S_{out}(n) : M'(s') = M(s') + \delta_{s_{out}=s'} \end{array}}$$

Example 4. *Given G from Example 3, we can compute our new marking using the evaluation rules. Incl can be applied at β, as both (α, β) and (γ, β) satisfy InclSearch. Assuming that $\mathit{eval}(C_{\beta\to\alpha})$, we can then generate a new marking*

$$M' = \{((\alpha,\beta),0),((\beta,\gamma),1),((\beta,\delta),0),((\gamma,\beta),0)\}$$

M' represents the new state of the process on which the token has been passed onward from the inclusive gateway.

3.3 Incremental and Distributed Implementations

The semantics given above formalizes the specification rather directly. If implemented naively, the computation of StepPathAnnotation and NonBlocking are a quite costly way to determine if an inclusive gateway can fire. Below we discuss two possible ways of implementing the semantics more efficiently by updating the data structures incrementally.

Incrementally computed path annotations. Alternatively to computing the path annotations from scratch for every step we can compute them incrementally. We then initially (and only once) compute the path annotations using the ComputePathAnnotation algorithm. Every time a gateway n is fired we update the path annotations for each flow s reachable from the outgoing flows of n as follows: If the marking of an outgoing s flow changes from 0 to 1 we remove n from the head of all the paths in annotations of the flows reachable from s (i.e. they receive a token and there was no token before). If all the incoming flows of the gateway n get marking 0 we also remove all paths from annotations of flows reachable from the outgoing flows that have n at their head. Figure 5 shows an example of this update on the example given in Figure 2 (default flows are left unspecified in the figure). After γ fires, it is removed from the heads of the paths of the downstream sequence flow annotations.

Precomputed path annotations and incrementally computed normalized markings. Given a marking, define the *normalized* marking $M_{0,1}$ by

$$M_{0,1}(s) = \delta_{M(s)>0}$$

For any BPMN_{inc} process there are finitely many $(2^{|S|})$ normalized markings.

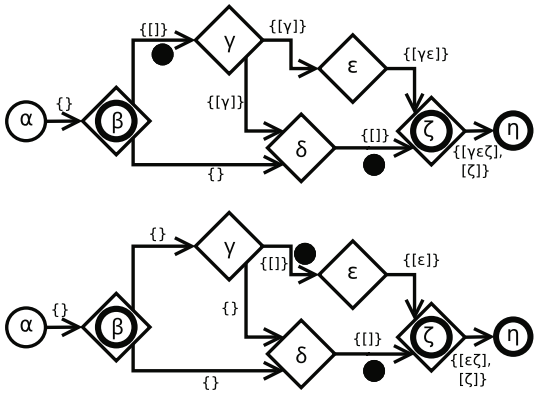


Fig. 5. Incremental path annotation update

Now, the function G computed by ComputePathAnnotations and the function *NonBlocking* for each flow only depend on the normalized marking of the flows that are reachable by following sequence flow backwards from the sequence flow in question. Consequently, as an alternative to incrementally maintaining the path annotations, the functions can be pre-computed for all of the possible normalized markings.

In addition to maintaining the markings of sequence flows, an implementation needs then only, for each sequence flow $(n, n') \in S$, to incrementally keep updated a table of normalized markings of the flows backwards reachable from the gateway n'. This can be done each time a gateway fires by updating the normalized marking (if it changes) for the flows reachable on paths starting on the outgoing flows of the gateway.

Distributed communication-based implementations. The two implementations above can be made in a distributed way by representing each gateway as a process and sequence flows as communication channels between processes. Each gateway process then maintains for each incoming flow the data structures (of respectively the path annotations or the table of normalized markings). It can then receive messages from gateways connected to incoming flows with updates, which can be forwarded (if necessary) to the gateways on outgoing flows.

4 Related Work

Previous work regarding the semantics of BPMN inclusive gateways can, broadly speaking, be divided into three main categories: semantics of BPMN that do not provide a non-local upstream search behavior for inclusive gateways, BPMN semantics that do, and semantics of other languages with similar constructs.

In [10], Völzer proposes three semantics for BPMN 2.0 inclusive gateways. Two of them are only restricted to the acyclic cases while the third semantics covers the general case (any kind of workflow) like ours. The latter is equipped with a linear algorithm for detecting the presence of upstream tokens.

All other previous work has been based on earlier versions of the BPMN standard, which were significantly less specific with regards to the semantics of the inclusive gateway. As far as the authors are aware, this is the first formal treatment of inclusive gateways making use of BPMN 2.0's semantics aside from [10], which introduced the approach used in the standard.

The definition of inclusive gateways in BPMN 1.0 is as follows:

> *Process flow SHALL continue when the signals (Tokens) arrive from all of the incoming Sequence Flow that are expecting a signal based on the upstream structure of the Process (e.g., an upstream Inclusive Decision).*

The particular deficiency of the specification of inclusive gateways in BPMN 1.0[4] is highlighted in a paper by Dijkman, *et al*, that provides a mapping from a large subset of BPMN to Petri nets, enabling the use of standard tools for analysis of processes.

Dijkman *et al* point out that the definition of inclusive gateways fails to specify which sequence flow should expect a signal, and especially does not take into account situations in which the inclusive gateway is upstream from itself. These concerns are largely solved in the draft version of BPMN 2.0, which is quite specific on these matters.

Wong and Gibbons[11] present a translation of a subset of BPMN process diagrams to CSP. They characterize the inclusive gateway as simply accepting tokens on some subset of the incoming sequence flows and then generating tokens on some subset of the outgoing sequence flows.

Prandi *et al* [8] define a mapping from BPMN to the process calculus COWS. Similarly to other mappings based on BPMN 1, an inclusive join is simply translated into a process that can receive a signal on any subset of its inputs and then sends a signal onwards.[8, p.256]

Ye, *et al*'s translation to YAWL [12] is perhaps the most faithful to BPMN in its implementation of inclusive gateways. The inclusive gateway is mapped to a YAWL OR-join. In YAWL, the OR-join has non-local semantics similar to those defined for BPMN's inclusive gateway [9]. Indeed, Dijkman suggested that the designers of BPMN borrow these semantics directly in an aside in [2].

The inclusive gateway semantics proposed in [1] is compatible with the vague phrasing of BPMN 1.0. However, the model adopted to deal with cyclical process graphs is not compatible with the better-specified semantics in [5], as it divides the tokens in the cycle into groups based on the iteration in which they are produced, and then considers each iteration separately. BPMN 2.0 [5] did not adopt this model, and it allows tokens from many different iterations to interact.

Dumas *et al.* [3] provide a semantics for BMPN's inclusive gateways based on the imprecise BPMN 1.0 specification, and their solution ends up very similar to ours. However, they provide a means for resolving the resulting deadlock in the vicious circle example, which is a situation in which two inclusive gateways depend on each other cyclically. Since the informal specification that was eventually adopted in BPMN 2.0 does not include this resolution strategy, and as our work is a faithful translation, we do not include it.

BPMN's inclusive gateway is quite similar to constructs called "OR-joins" in various other process formalisms, in particular YAWL[9] and EPCs. Indeed, Dijkman suggested that the designers of BPMN borrow YAWL's semantics directly in an aside in [2].

In [6], Kindler points out that the informal semantics of OR-joins in Event driven Process Chains (EPCs), which contain a similar non-local backwards condition, can not be formalized consistently, due to the same sort of vicious circle treated in [3]. The BPMN 2.0. specification eliminates this problem, although the vicious circle example will result in a deadlock. However, as illustrated by the example in Fig. 2 BPMN 2.0 may exhibit race-conditions.

The primary difference between BPMN inclusive gateways and OR-joins in other notations is that a token that is upstream of both an empty and a full incoming sequence flow does not block the activation of the gateway in BPMN, while other languages do not include this stipulation. Therefore, straightforward translations of inclusive gateways to OR-joins such as Ye, *et al*'s translation to YAWL[12] are insufficient to capture BPMN 2.0's semantics.

5 Conclusion and Future Work

The particular behavior defined for inclusive gateways in BPMN 2.0 makes it difficult to provide a local semantics. However, given access to global information about the current state of execution, a precise semantics for inclusive gateways can be provided.

In this paper, one such precise semantics is provided. It differs from other approaches in that it does not attempt to translate BPMN process diagrams into other, better-understood calculi or process modeling languages. Instead, the semantics is provided directly in terms of a subset of BPMN. While this approach precludes the use of tools and methods developed for these other models, it can allow use of the semantics more easily either in implementations of BPMN or in providing analyses that are more easily understood by less-technical users who may not be familiar with other process models. Due to the non-local nature of inclusive gateways, this semantics requires the use of a backwards search algorithm for determining the parts of the global execution state that are relevant to each inclusive gateway at each step of the execution. We have sketched in Section 3.3 two approaches to how this search can be replaced by incremental updates to respectively token-paths annotations and to local copies of (normalized) markings after a gateway is fired.

Possibilities for future work include extending the semantics to cover more of BPMN, formalizing the incremental implementations of the semantics and prove their correctnes, and investigating applications of the semantics to real projects that make use of BPMN 2.0. Additionally, other approaches than token-based semantics, such as graph rewriting, can possibly be developed to give a simpler semantics for inclusive gateways. We also plan to implement the semantics following the different approaches sketched in Section 3.3, analyze their complexity and compare the implementations using a set of example BPMN_{inc} processes.

References

1. Börger, E., Sörensen, O., Thalheim, B.: On defining the behavior of OR-joins in business process models. J. UCS 15(1), 3–32 (2009)
2. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information and Software Technology 50(12), 1281–1294 (2008)
3. Dumas, M., Großkopf, A., Hettel, T., Wynn, M.T.: Semantics of standard process models with OR-joins. In: Chung, S. (ed.) OTM 2007, Part I. LNCS, vol. 4803, pp. 41–58. Springer, Heidelberg (2007)
4. Object Management Group. BPMN 1.0: OMG final adopted specification (February 2006), `http://www.bpmn.org/Documents/OMG_Final_Adopted_BPM_1-0_Spec_06-02-01.pdf` (accessed May 10, 2010)
5. Object Management Group. Business process modeling notation (BPMN) 2.0 beta 1(August 2009), `http://www.omg.org/cgi-bin/doc?dtc/09-08-14.pdf` (accessed May 10, 2010)
6. Kindler, E.: On the semantics of EPCs: Resolving the vicious circle. Data Knowl. Eng. 56, 23–40 (2006)
7. Peterson, J.L.: Petri nets. ACM Computing Surveys 9(3), 223–252 (1977)
8. Prandi, D., Quaglia, P., Zannone, N.: Formal analysis of BPMN via a translation into COWS. In: Wang, A.H., Tennenholtz, M. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 249–263. Springer, Heidelberg (2008)
9. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
10. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 294–309. Springer, Heidelberg (2010)
11. Wong, P.Y.H., Gibbons, J.: A process semantics for BPMN. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008)
12. Ye, J., Sun, S., Song, W., Wen, L.: Formal semantics of BPMN process models using YAWL. In: Second International Symposium on Intelligent Information Technology Application, IITA 2008, vol. 2, pp. 70–74, 20-22 (2008)

Failure Analysis for Composition of Web Services Represented as Labeled Transition Systems*

Dinanath Nadkarni, Samik Basu, Vasant Honavar, and Robyn Lutz

Department of Computer Science, Iowa State University, Ames, IA 50011, USA
{yogesh,sbasu,honavar,rlutz}@cs.iastate.edu

Abstract. The Web service composition problem involves the creation of a choreographer that provides the interaction between a set of component services to realize a goal service. Several methods have been proposed and developed to address this problem. In this paper, we consider those scenarios where the composition process may fail due to incomplete specification of goal service requirements or due to the fact that the user is unaware of the functionality provided by the existing component services. In such cases, it is desirable to have a composition algorithm that can provide feedback to the user regarding the cause of failure in the composition process. Such feedback will help guide the user to reformulate the goal service and iterate the composition process. We propose a failure analysis technique for composition algorithms that views Web service behavior as multiple sequences of input/output events. Our technique identifies the possible cause of composition failure and suggests possible recovery options to the user. We discuss our technique using a simple e-Library Web service in the context of the MoSCoE Web service composition framework.

1 Introduction

A number of formal approaches [DS05, HS04, tBBG07] have been developed in recent years to address the problem of service composition. These approaches take as input the specification of existing service functionalities and the desired functionality (also referred to as the *goal*) in a specific formalism, and automatically generate a choreographer that mediates the communication between a subset of existing services to realize the goal (if possible). In addition to automation, these approaches also provide formal guarantees of the correctness of the composition.

Typically, the existing approaches can be viewed as a single-step process, where the result is either a feasible composite service or no result at all when the composition process fails to generate a composite service that conforms to the goal functionality. We claim that such failures may be due to the fact that

* This work is supported in part by NSF grant CCF0702758.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 161–175, 2011.

the developer may not be aware of all the details of existing services' functionalities and as a result s/he may specify certain goal functionality that is impossible to realize using any of the existing services. However, if the developer were provided with some feedback and/or suggestions regarding the cause of composition failure, then the developer would be able to reformulate the goal functionality without violating the overall desired requirements such that the new goal would become realizable from the composition of existing services. Such a process may be iterative, resulting from multiple composition failures, failure analysis and re-formulations.

In this context, we propose methods to analyze the cause of composition failures and to provide feedback to the developers based on the analysis. We consider the problem in the MoSCoE service composition framework [PBLH06], where services and the goal functionalities are described as labeled transition systems. States in the transition system represent the configurations of the service/goal and transitions labeled with input/output events represent how the service evolves from one configuration to another. The composition algorithm in MoSCoE aims to identify the communication pattern between existing services via a choreographer such that the resulting transition system describing the composite service mimics every behavior of the goal service. Failure to generate a composite service in MoSCoE, therefore, is due to the existence of transitions in the goal that cannot be replicated by any composition of the existing services. This, in turn, implies that the given input sequence as specified by the goal functionality is not sufficient to produce the required output sequences. Once our method identifies the cause of the failure, it suggests possible changes to the goal transition system that can address the failure and lead to a successful composition. The developer can then choose to incorporate the suggestions and re-run the composition process.

The rest of the paper is organized as follows. Section 2 presents an illustrative example that will be used in the rest of the paper to explain the salient aspects of our work. Section 3 provides a brief overview of the MoSCoE composition algorithm. Section 4 discusses the various scenarios that can cause the failure of the composition followed by our method to identify them. Section 5 discusses the application of our method on the illustrative example. Section 6 gives a summary of our work and describes future avenues of research.

2 Illustrative Example

Consider a library book reservation service (`eLibrary`) that requires three main functionalities: book searches, book delivery requests and book reservations. The goal of the service is to allow a library member to search through the library catalog for a book based on parameters such as the book title and the author. If the library has copies of the book, the service checks if a copy is available to be checked out. If it is, the service places a request for delivery of the book to the member's home address, which is stored in the member's account information. If all copies of the book have been checked out, the service places a hold request on

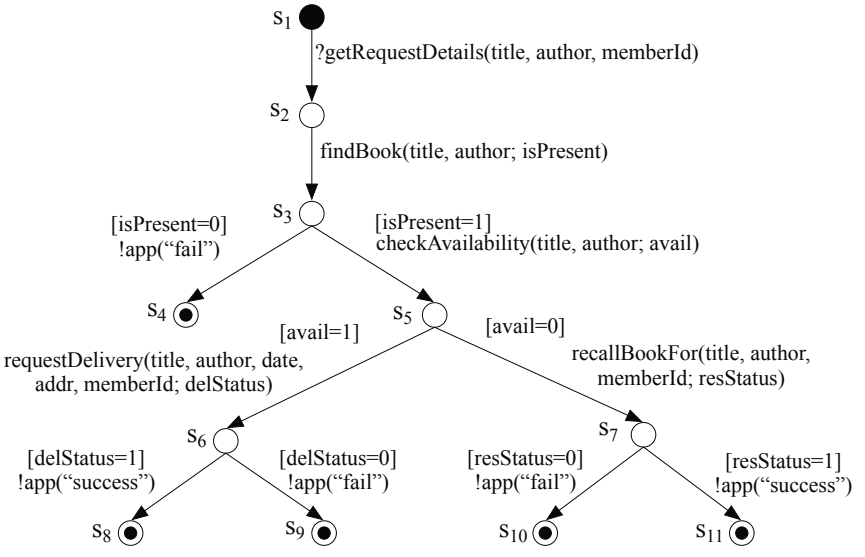
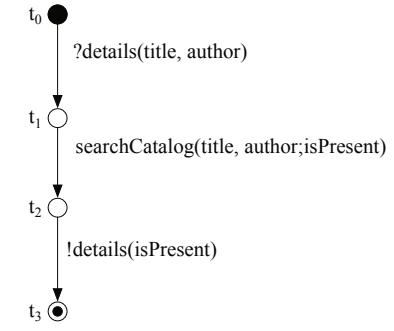


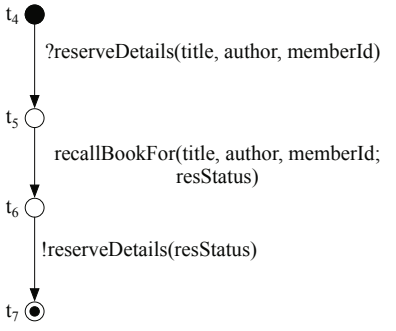
Fig. 1. Specification of goal service `eLibrary` as labeled transition system

the book. The Web service developer is assigned the task of generating the above service. The developer prepares the transition system (Figure 1) representing the goal behavior. There are three types of transitions: input, denoted by `?`, output, denoted by `!`; and function invocation. For instance $s_1 \longrightarrow s_2$ is an input transition, $s_3 \longrightarrow s_4$ is an output transition and $s_2 \longrightarrow s_3$ is a function invocation. Input/output transitions contain a message header and a message body, e.g., `getRequestDetails` is the message header and its parameters denote the message body. Transitions denoting function invocation contain the function name, the input parameters and the output of the function. For instance, `findBook` is the function name, `title` and `author` are input parameters and `isPresent` is the output. Transitions may also contain a guard (enclosed in `[·]`) denoting the condition under which the transition is enabled; transitions with no guards are always enabled.

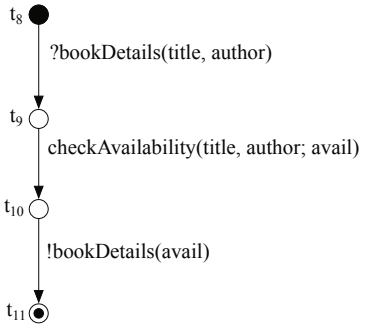
The repository that will be used to generate the composite service contains the following services: `Availability`, `BookReservation`, `DeliveryRequest`, `MemberAddress` and `SearchBook`. The `Availability` service accepts as input the title of a book and checks whether a copy of the book can be checked out. The `BookReservation` service accepts as input the book title and the member ID, then places a recall request for a copy of the book for that member. `DeliveryRequest` places a request for delivery of a book to the address specified in the member's account on a particular date. `MemberAddress` accesses the member's account details and returns the member's home address. Finally, the `SearchBook` service searches the library catalog for a book given the title and the name of the author. The labeled transition system specifications of each of these services are shown in Figure 2. The objective of the composition algorithm



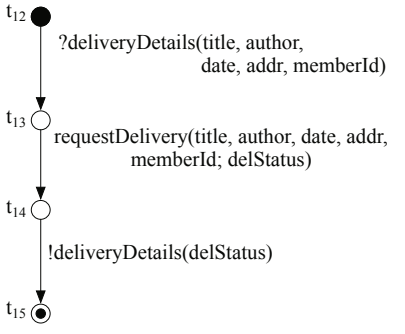
(a) `SearchBook`



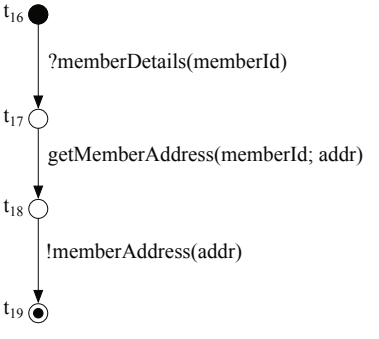
(b) `BookReservation`



(c) `Availability`



(d) `DeliveryRequest`



(e) `MemberAddress`

Fig. 2. Component services

is to generate a choreographer that will mediate the communication between the component services such that the behavior of the component services and the choreographer replicates the behavior of the goal service. Note that the choreographer cannot generate any messages and cannot provide any functions.

We consider this simple example scenario to explain the salient aspects of the proposed method. It is worth mentioning that though the existing services in the example do not contain any loops and branching-behavior, the composition algorithm we considered (MoSCoE, see Section 3) and the proposed failure analysis based on MoSCoE (see Section 4) are capable of handling services with loops and branches.

3 MoSCoE Composition Algorithm

3.1 Web Services as Transition Systems

Definition 1 (Service Transition System (STS)). *A Web service transition system is a tuple $W = (S, s_0, F, T)$ where S is the set of states, $s_0 \in S$ is the start state, $F \subseteq S$ is the set of final states and T is the set of transitions between pairs of states. A transition is of the form $s \xrightarrow{g,a} s'$ where $s \in S$ and $s' \in S$ are source and destination states of the transition, g is the guard on the transition and a is the event that is executed by the transition. There are four types of events:*

- *input events denoted by* `?msgHeader(msgBody)`,
- *output events denoted by* `!msgHeader(msgBody)`, *and*
- *function invocation denoted by* `functionName(InputParameters; Output)`.
- *internal event denoted by* τ.

Transitions are referred to as input, output, function, or internal based on their labels. An Internal event denotes computation or communication performed by a (composite) service that is not observable to the client.

Example 1. Figures 1 and 2 illustrate STS-representations of goal and existing services. The state states are represented by $\bullet$ and the final states are represented by $\circledcirc$. In Figure 1, the transition $s_1 \longrightarrow s_2$ is an input transition with *true* guard; the transition $s_2 \longrightarrow s_3$ is a function invocation with the name of the function `findBook` and with *true* guard; the transition $s_3 \longrightarrow s_4$ is an output transition with a guard on `isPresent`.

Definition 2 (Parallel Composition of STS [PBLH06]). *Given two STSs $W_1 = (S_1, s_{01}, F_1, T_1)$ and $W_2 = (S_2, s_{02}, F_2, T_2)$, their parallel composition under the restriction set L, denoted by $(W_1 \| W_2) \backslash L$, is a tuple (S, s_0, F, T) where $S \subseteq S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$, $F \subseteq F_1 \times F_2$ and T is the transition relation described as follows: for $(i, j) \in \{(1, 2), (2, 1)\}$*

1. $s \xrightarrow{g_1, ?m(\boldsymbol{x})} s' \in T_i \ \wedge \ t \xrightarrow{g_2, !m(\boldsymbol{x})} t' \in T_j \ \wedge \ m \in L \Rightarrow (s, t) \xrightarrow{g_1 \wedge g_2, \tau} (s', t')) \in T$
2. $s \xrightarrow{g_i, e} s' \in T_i \ \wedge \ \boldsymbol{header}(e) \notin L \Rightarrow (s, t) \xrightarrow{g_i, e} s'(s', t) \in T.$

where

$$\boldsymbol{header}(e) = \begin{cases} m \text{ if } e \in \{?m(\boldsymbol{x}), !m(\boldsymbol{x})\} \\ \perp \text{ otherwise} \end{cases}$$

The parallel composition describes the rules by which two or more services can communicate with each other. The first rule in the transition relation describes a *synchronous* move where one services provides an output that is consumed as input by the other service. The result is an internal transition in the composite service. The second rule, on the other hand, is an autonomous move by the individual services. These rules are derived from CCS-style synchronization in process algebra [Mil82].

3.2 The Service Composition Problem

Given the transition system of a goal, W_g, and the set of available components $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$, the problem of service composition entails identifying a choreographer transition system W_c such that

$$((W_{i1} \| W_{i2} \| \ldots \| W_{ik}) \| W_c) \setminus L \approx W_g \tag{1}$$

where L is the set of actions that are not present in W_g and $\approx$ is the largest relation describing *weak bisimilarity* [Mil82] between pairs of states. This definition ensures that the goal W_g and the composite service containing $W_{i1}, \ldots, W_{ik}$ and W_c exhibit observable behaviors that are temporally indistinguishable (i.e., no temporal logic can differentiate between the behaviors). Note that, the weak bisimulation is only concerned with the observable events, i.e, all internal and unobservable behaviors (events involving τ) are ignored. The choreographer generated by the composition algorithm can be viewed as a new service with the restriction that it cannot have any function invocation or internal event; the role of the choreographer is to buffer and relay messages between existing services.

The composition algorithm takes as input the start states of the existing component service transition systems and that of the goal transition system and iteratively performs the following computations:

0. a transition in the goal is enabled only when the variables in the corresponding transition-guard are available at the current state of the choreographer;
1. if the transition in the goal is an input, then generate the corresponding input transition in the choreographer, and move the goal transition system to the next state;
2. if the transition in the goal is an output and the output messages are available to the choreographer, then generate the corresponding output transition in the choreographer, and move the goal transition system to the next state;
3. if the transition in the goal is a function and there exists a service W_i that can supply the function transition from its current state, then move W_i and the goal transition state to their corresponding next states;
4. if the transition in the goal is a function and there exists a service that can supply the function transition from some future state, then identify any service that can make a move on some input available to the current state of the choreographer and move the choreographer and the corresponding service to their next states.

Consider the illustrative example in Figures 1 and 2 (Section 2). The transition in the goal $s_1 \longrightarrow s_2$ is an input transition—this indicates that the goal service expects input from the client (who will use the service) to move from s_1 to s_2. Therefore, following Rule 1 above, the transition is replicated in the choreographer, which will act as the interface between the client and the existing services. A pair of new states c_1 and c_2 are created for the choreographer such that $c_1 \longrightarrow c_2$ is labeled by the input event that labels $s_1 \longrightarrow s_2$. At state c_2, the messages `title`, `author` and `memberId` are available to the choreographer, as these are supplied by the client. If the goal is at state s_3, for the choreographer to replicate either of the transitions $s_3 \longrightarrow s_4$ and $s_3 \longrightarrow s_5$, the choreographer needs to be at a state where `isPresent` is available to the choreographer (see Rule 0 above).

If after repeated applications of the rules described above, the goal moves to the final state, then the parallel composition of the generated choreographer and the existing services is weakly bisimilar (see Equation 1 in Section 3.2) to the given goal (*soundness*). On the other hand, if none of the above rules for choreographer generation are applicable, then the composition process fails and there exists no choreographer that can realize the given goal (*completeness*). For details of the described method and the proof of its soundness and completeness, refer to [PBLH06].

The above method is described in the context of the MoSCoE service composition framework. Similar methods that also represent services and the goal using transition systems are developed by [BCD+05, CDL+08, HB03].

4 Failure Analysis for MoSCoE Composition

In this paper, we focus on the cases where the service composition process fails, i.e., during the iterative choreographer generation process described in Section 3 the goal, the generated choreographer and the existing services move to states from which none of the rules can be applied. We augment the composition algorithm with a method which identifies the cause of the failure of the composition process and provides feedback/suggestions (to avoid failure) to the developer.

The feedback given to the developer is of utmost importance as it allows for progressive service composition. The feedback provides information not only regarding the cause of the composition failure but also as to how such failure can be resolved. Two problems need to be addressed for developing a feedback process that can be effectively used in practice: (a) what is to be given as feedback and (b) how that feedback is to be presented to the developer.

4.1 Tree of Recovery Options

When failure occurs during composition, the cause of the failure is identified. Failures occur mostly due to missing messages that are required as input to functions or input transitions to some service, or due to functions required by the goal and not provided by any of the existing services. Such failures are resolved

by finding alternate paths in the component services, by calling on components that provide the missing message sets, or by identifying a semantically equivalent function in the existing services. In short, multiple resolutions are possible for each failure.

The failure analysis approach presented in this work explores each such recovery option. Upon failure, all possible recovery options are identified and a *choice point* is created in the composition computation. For each recovery option, a new branch in computation is explored using the corresponding option. In every branch, the goal service is modified based on the recovery solution corresponding to that branch. Once the goal service has been modified, the composition process continues using the modified goal service. If one or more branches (at each choice point), leading to a different modification to the original goal service, eventually terminate successfully, the developer is provided with the information regarding the various goal service modifications. The developer can decide to either select one of these modifications or discard all modifications and reformulate the goal from scratch. We refer to the above computation process as the *recovery tree*.

A sample computation tree is illustrated in Figure 3. The composition process starts and fails for the first time at node n_1. Three branches are created from this node, with each branch representing one solution to the failure at node n_1. Two of these solutions fail during the simulation, but node n_3 indicates that the modified goal service corresponding to the second solution is successfully composed. There are two possible solutions to the failure at node n_2 and both of them are explored, as shown. Node n_8 indicates that the goal service modified at node n_6 results in failure. This branch is marked as failed; for the purpose of efficiency, we do not consider exploring any paths that contain more than three choice points. This is because the modifications deployed to the goal service in multiple choice points are likely to make the modified goal service excessively different from the original developer-specified goal service and as such are likely to be discarded by the developer.

4.2 Identifying Recovery Options

The following cases would result in failures during the composition process:

1. A guard condition cannot be examined as variables required for the condition are not available
2. Messages for an output transition are not available to the choreographer
3. Messages for an input transition to an existing service are not available to the choreographer
4. A required function cannot be invoked, that is, none of the existing services provide a function specified in the goal service

In the following, we discuss the recovery options identified for each of the above failure scenarios.

I: Failure due to guard conditions. This can happen when a variable (either provided via input from the client or output from some existing service) has not

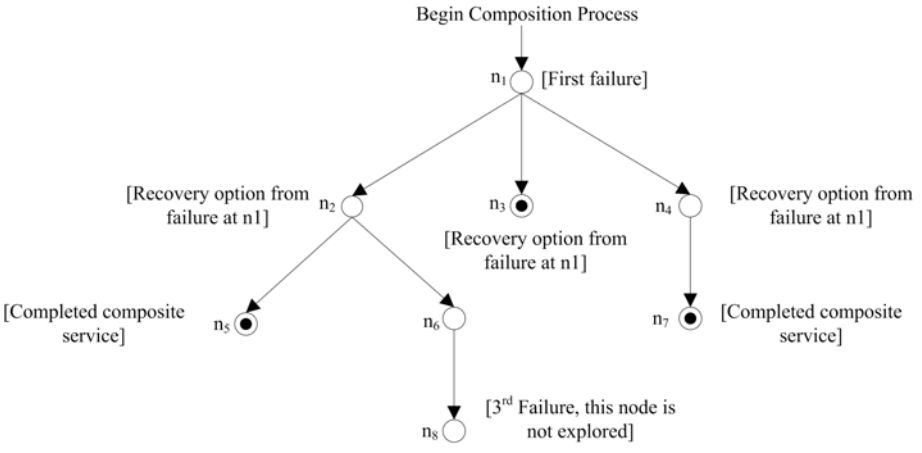


Fig. 3. Recovery Tree

yet been available to the generated choreographer, and the composition process encounters a guard on such a variable in the goal transition. There are two possible recovery options from this failure based on the following:

- There exists a service that has an output containing the missing variable(s) in the output message body. The recovery option is to identify the input that must be provided at the current state of this service such that eventually the required output can be obtained.
- If no such service exists, the goal service is modified to include an input transition requesting the client to provide the message that is required for the guard condition.

II: Failure due to unavailability of output message. In this scenario, the composite service has to provide an output message to the client but is unable to do so as it does not have the output message set. This happens when the algorithm encounters an output transition in the goal service and the choreographer store does not have the output message for such a transition. Recovery from this failure is based on the existence of a service that performs the required output action as specified by the goal. The recovery option is to identify the input that must be provided at the current state of this service such that eventually the required output can be obtained.

III: Failure due to unavailability of input message. In this scenario, the choreographer has to provide an input message to the component, which might fail if the choreographer does not have the required message set. This scenario also includes the case where a function is to be called and the message set required to call the function is not available. The recovery option in this case is the same as scenario I above.

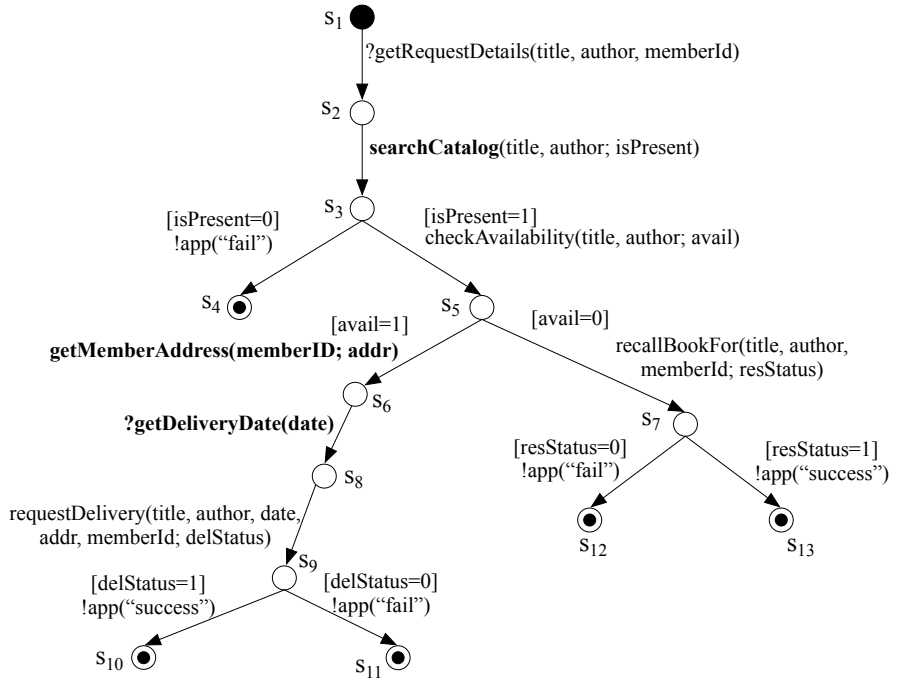


Fig. 4. Suggested Goal; modifications suggested as part of recovery from composition failure are shown in **bold**

IV: Failure due to unavailability of required function. In this case, the goal service has a function invocation that is not provided by any of the existing services. To recover from this failure, we search for a semantically equivalent function with the same input and output messages as the required function. The transition on the missing function in the goal is then replaced by a transition on this semantically equivalent function.

5 Case Study

We discuss the application of the failure identification and recovery options using the illustrative example introduced in Section 2. The objective is to solve the following problem

$$\exists W_c : (\texttt{SearchBook} \| \texttt{BookReservation} \| \texttt{Availability} \| \texttt{DeliveryRequest} \| \texttt{MemberAddress}) \| W_c) \backslash L \approx \texttt{eLibrary}$$

and identify a W_c (choreographer, if one exists). In the above, the restriction set $L = \{m \mid m = \texttt{header}(e) \wedge e \in \text{existing services}\}$, i.e., L contains the header of events on which the existing services can communicate with the generated choreographer (see Section 3.2).

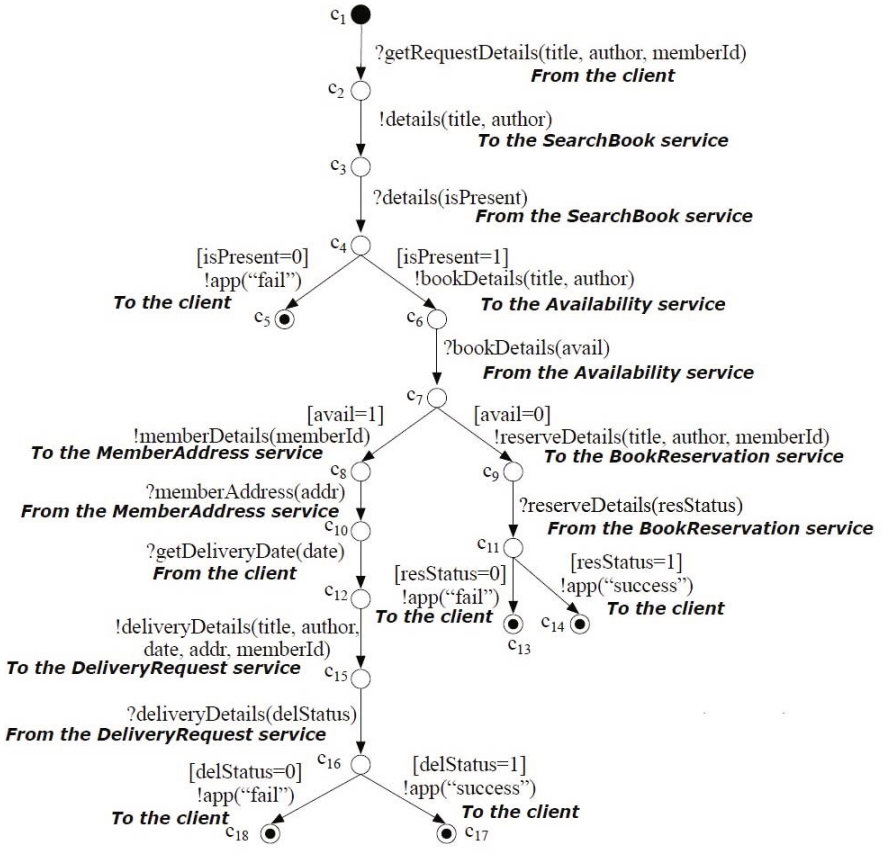


Fig. 5. Generated Choreographer for the modified Goal (Figure 4); transitions of the choreographer are annotated with the service or client (in **bold**) which participates in communication via the event on the transition

The composition algorithm takes as input the goal as specified by the developer and the set of component services, and attempts to create a choreographer for the eLibrary system. The input action `?getRequestDetail(title, author, memberId)` in the goal corresponds to the receipt of a message from the client, meaning that the client has entered data in the system. The choreographer mimics this input action and stores the message body in the choreographer message store.

The next step in the composition is to create a transition that realizes the function invocation `findBook` (see Figure 1). The composition process fails as none of the component services can provide the required invocation (see scenario IV in Section 4.2). The failure analysis method identifies the `searchCatalog` function provided by the `searchBook` service as a possible replacement for the `findBook` function, based on matching input parameters and output. The replacement function should be compared with the original for semantic equivalence. Checking

a function for semantic equivalence[1] can be done based on the type of the inputs and outputs associated with the function (see for example, [Bur04, BHS03, DMD+03]). Based on this recovery option, a choice point and a branch are created in the recovery tree. The branch considers a modified version of the goal service where the function `findBook` is replaced by `searchCatalog` (see Figure 4). All the computation along this branch uses this modified goal as input.

Note that the `searchBook` service is at state t_0. In order to replicate the function invocation `searchCatalog` in the modified goal service, it is necessary to move the `searchBook` service to state t_1. In order to realize this, the choreographer is required to provide the input to the `searchBook` service at state t_0. Therefore, a transition on the input action `!details(title, author)` (to be consumed by the `searchBook` service with the same message header) is created in the choreographer (see Figure 5); all the elements in the message body (i.e., `title` and `author`) are available to the choreographer (i.e., present in the choreographer store).

Composition proceeds without any failure and the choreographer is generated to communicate with the `Availability` service such that the invocation of the function `checkAvailability(title, author; avail)`, as prescribed in the goal, can be realized. However, when the goal-state s_5 is reached, the composition process fails to replicate the transition $s_5 \longrightarrow s_6$. This is because the choreographer cannot provide `addr` information as required by the input to the function `requestDelivery(title, author, addr, memberId; delStatus)`. This corresponds to scenario III described in Section 4.2. To recover from this failure, our method searches for an existing service that has an output transition with message body containing `addr`. Such a service is identified to be `memberAddress`. A choice point and the corresponding branch of computation are created and it is given a copy of the goal service, the choreographer and the component services. The input variables required to initiate the `memberAddress` service are available to the choreographer, so the recovery method simply inserts a transition on `getMemberAddress` in the goal service for the new branch.

To support this function invocation in the modified goal service, two new transitions on the output and input actions, `!memberDetails(memberId)` and `?memberAddress(addr)`, are created in the choreographer. At this point, the choreographer store has all the required inputs of `requestDelivery` function, except `date`. As a result, the composition process fails again.

The failure analyzer searches for any service that can provide an output `date`. As no such service exists, the only recovery option is to insert an input operation in the goal that requests the client to provide the `date`.

Since the entire message set for the `requestDelivery` function is now available, this function can now be invoked. The algorithm supports this invocation by creating two new actions `!deliveryDetails(date, addr, memberId)` and `?deliveryDetails(delStatus)` in the suggested choreographer. This means that an output message `!deliveryDetails(date, addr, memberId)` was sent

[1] Methods for identifying semantically equivalent functions are beyond the scope of this work.

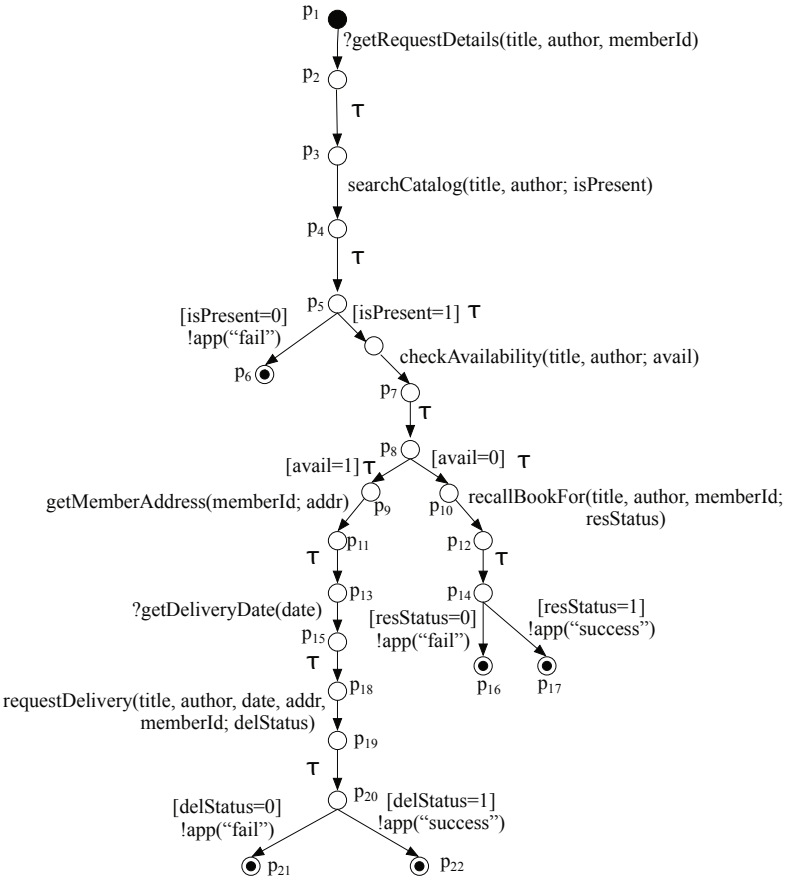


Fig. 6. Composition of generated choreographer (Figure 5) and existing services (Figure 2); composite service is weakly bisimilar to suggested goal (Figure 4)

to the `memberAddress` service, the function `getMemberAddress` was invoked and the output of the function call was then sent back to the choreographer. The composition process, finally, considers the behavior of the goal as specified by transitions from $s_5 \longrightarrow s_7 \longrightarrow \ldots$. The composition process successfully completes generating the choreographer without failure.

Figure 4 shows the suggested goal following the recovery options described above; the modifications are shown in bold. Figure 5 illustrates the corresponding choreographer generated by the composition process. Figure 6 shows the composite service that will be realized when the generated choreographer is composed with the existing service. It can be shown that the composite service is (weak) bisimulation equivalent to the suggested goal-transition system.

6 Conclusion

The failure analysis and the recovery techniques proposed in this work help in identifying the cause of failure in the composition process and provide appropriate feedback to the developer. The feedback is described as possible modifications to the goal service for every possible recovery from the failure. Though the technique is described in the context of MoSCoE service composition framework [PBLH06], it can be applied with minimal modification, to the analysis of failures of the composition process where the composition is defined over different variations of labeled transition systems.

As part of future work, we plan to investigate the efficiency and applicability of the proposed method in practical settings using real-world and benchmark service composition problems. We will also work to develop failure analysis techniques where the goal is specified in the language of temporal logic (e.g., EAGLE [PTB05]), description logic([BCG+03]), etc., and the composition problem is reduced to the satisfaction problem (instead of the equivalence problem as described in MoSCoE).

References

[BCD+05] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic Service Composition based on Behavioral Descriptions. Intl. Journal on Cooperative Information Systems 14(4), 333–376 (2005)

[BCG+03] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: e-service composition by description logics based reasoning. In: Calvanese, D., De Giacomo, G., Franconi, E. (eds.) CEUR Workshop Proceedings of Description Logics, vol. 81 (2003), CEUR-WS.org

[BHS03] Baader, F., Horrocks, I., Sattler, U.: Description logics as ontology languages for the semantic web. In: Festschrift in honor of Jörg Siekmann. LNCS (LNAI), Springer, Heidelberg (2003)

[Bur04] Burstein, M.: Dynamic Invocation of Semantic Web Services that use Unfamiliar Ontologies. IEEE Intelligent Systems 19(4), 67–73 (2004)

[CDL+08] Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M., Patrizi, F.: Automatic service composition and synthesis: the Roman model. IEEE Data Eng. Bull. 31(3), 18–22 (2008)

[DMD+03] Doan, A., Madhavan, J., Dhamankar, R., Domingos, P., Halevy, Ã.: Learning to Match Ontologies on the Semantic Web. VLDB Journal 12(4), 303–319 (2003)

[DS05] Dustdar, S., Schreiner, W.: A Survey on Web Services Composition. International Journal on Web and Grid Services 1(1), 1–30 (2005)

[HB03] Hamadi, R., Benatallah, B.: A Petri Net-based Model for Web Service Composition. In: 14th Australasian Database Conference, pp. 191–200. Australian Computer Society, Inc. (2003)

[HS04] Hull, R., Su, J.: Tools for Design of Composite Web Services. In: ACM SIGMOD Intl. Conference on Management of Data, pp. 958–961 (2004)

[Mil82] Milner, R.: A Calculus of Communicating Systems. Springer, New York (1982)

[PBLH06] Pathak, J., Basu, S., Lutz, R., Honavar, V.: Parallel Web Service Composition in MoSCoE: A Choreography-Based Approach. In: 4th IEEE European Conference on Web Services, pp. 3–12. IEEE CS Press, Los Alamitos (2006)

[PTB05] Pistore, M., Traverso, P., Bertoli, P.: Automated Composition of Web Services by Planning in Asynchronous Domains. In: 15th Intl. Conference on Automated Planning and Scheduling, pp. 2–11 (2005)

[tBBG07] ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Web service composition approaches: From industrial standards to formal methods. In: ICIW, p. 15. IEEE Computer Society, Los Alamitos (2007)

On Nondeterministic Workflow Executions*

Alexandra Potapova and Jianwen Su

Department of Computer Science
University of California, Santa Barbara
{a.potapova, su}@cs.ucsb.edu

Abstract. The ability to compose existing services to form new functionality is one of the most promising ideas enabled by SOA and the framework of (web) services. A composition or a workflow often involves services distributed over a network and possibly many organizations and administrative domains. Nondeterminism could occur in a composition in at least two ways. The first form is the result of modeling abstraction that hides the detail information and thus makes the "computation" appear non-deterministic. The second form is closely related to "operational optimization", e.g., one may try to invoke more than multiple services for a task, whichever completes first will produce the result and preempts all other services. In this paper, we focus on the latter and measure the complexity of service execution as the amount of needed resources and controlling mechanism for executing nondeterministic service compositions. We formalize the model and complexity problem and develop technical results for this problem in the general setting as well as special cases.

1 Introduction

Web services are bite-sized pieces of software components that can be executed over, e.g., the Internet (see [9]). The ability to organize, manage, discover, compose, and invoke web services creates a great potential in changing the way we develop software systems for applications [10,16]. To a degree, this change is already happening, e.g., in workflow management [17]. An immediate goal is to be able to dynamically share web services in a similar manner to the sharing of data on the Web. This includes, in particular, composing geographically distributed web services. Nondeterministic web services are an important class of web services that have been studied variously [19,15,5,1,20,3,6,11]. Business processes or workflows are probably the most frequently seen web service compositions in applications. This paper studies a new and interesting management problem for executing nondeterministic workflows.

Generally, nondeterminism in services/workflows has several possible causes. One is modeling abstraction. Indeed, hiding data values may turn a conditional choice into a nondeterministic choice. Nondeterministic web services and workflows in many early studies are in this category [5,1,3]. Another is optimization. For example, if we have two candidate tasks (to reserve a ticket) to execute but only one (ticket) is needed, we may proceed to invoke both; whichever completes first would produce the result and should end the execution (of both). In some cases, such an opportunistic strategy is needed due

* Support in part by NSF grants IIS-0812578 and a grant from IBM.

M. Bravetti and T. Bultan (Eds.): WS-FM 2010, LNCS 6551, pp. 176–190, 2011.

to "deferred choice" [6]. Also, collaborative processes are often a contributing factor for nondeterminism [19,15,11].

If services/workflow are truly nondeterministic (e.g., reasons other than the first listed in the above), managing executions of nondeterministic services/workflows needs additional support [20]. In this paper we focus on the problem of keeping track of all threads of nondeterministic executions and formulate a new notion of "degree of nondeterminism" to measure a type of difficulty in execution management. We study the decision and computation problems associated with this notion.

Consider a workflow (or composite service) represented as a nondeterministic finite state machine where states are *geographically distributed*. (The assumption on states is a direct consequence of invoking distributed geographically services and control flow being tightly associated with invocation.) Executing the workflow would mean to follow all possible nondeterministic execution paths simultaneously. For example, if the current state has two possible next states after an activity, both would need to continue the execution since we do not know which path will lead to a (successful) completion. If the (graph of the) workflow is acyclic, there is clearly a bound on the number of execution paths to manage (assuming each service takes a unit time) that depends on the structure of the state machine. If the workflow has cycles, it is unclear whether such a bound exists. The central problem studied in this paper is to decide if such a bound exists and compute the bound if it does.

In this paper we investigate the upper bound on the number of execution paths in workflows under two models. The first model is called "organization flows". An organization flow is a graph with an initial node and several final nodes, and each outgoing edge is interpreted as a possible nondeterministic execution. Such workflows correspond to structures of organizations that can execute workflows. The second model extends the first by adding edge labels (tasks) and assigning (geographical) locations to states. We study two subclasses of such workflows, "fully distributed" where each state is assigned a distinct location, and "transformation flow" where all states are assigned to a fixed location. We present the following technical results:

1. We show that boundedness of degree of nondeterminism for arbitrary organization flows can always be decided in linear time, and for a subclass of organization flows the degree can be computed in cubic time.
2. For fully distributed workflows, both boundedness and computation of nondeterminism degree can be done in exponential space and time, respectively.
3. For transformation flows, we can construct equivalent workflows with a minimum degree (i.e., the state machine is deterministic).

This paper is organized as follows. Section 2 focuses on organization flows and their nondeterminism. Section 3 defines a general model for workflows, generalizes the nondeterminism notion, and presents technical results on fully distributed workflows; transformation flows are presented in Section 4. Section 5 concludes the paper.

2 Organization Flow

In this section, we study "organization flow," which focuses on organizations of administration domains and their connectivity in supporting workflows.

(a) A city office (Π_1) (b) A travel agency (Π_2)

Fig. 1. Examples of Organization Flows

Example 2.1. As shown in Fig. 1(a), a city office [12] for handling real estate permits consists of 5 stations: receptionist (R), preliminary (P) and secondary (S) reviews and decisions, manager (M), and document archival (A). Typically, the receptionist decides to accept or deny an application. The manager approves cases, or could decide to reconsider a case. Fig. 1(a) shows all five stations and possible case trails between them. Note that all case documents are electronic, there is no requirement that a case is only at one station at a time.

Similarly, Fig. 1(b) shows a travel agency where the front desk (F) collects requirements, an agent plans the itinerary (I) by consulting with an external agency (B) for booking individual segments, a cashier (C) handles payments from the customer and to other agencies, and the completed plan is delivered (D) to the customer. ∎

In the technical discussions, let $\boldsymbol{L}$ be an infinite set of *locations*.

Definition. An *organization flow* is a tuple (V, E, s, F) where $V \subseteq \boldsymbol{L}$ is a finite set of locations, $s \in V$ is the *initial* location, $F \subseteq V$ is a set of *terminal* (or *final*) locations, and $E \subseteq (V-F)\times V$ is a set of (directed) edges over V.

Definition. Let $n \geqslant 0$ be an integer and $\Pi = (V, E, s, F)$ an organization flow. An *enactment of length n over Π* is a sequence of locations $\pi = v_0 v_1 \cdots v_n$ where v_i's are locations in V, $v_0 = s$ (the initial location), and for each $i \in [0..(n-1)]$, $(v_i, v_{i+1}) \in E$. The enactment π is *complete* if v_n is in F (terminal).

Fig. 1 shows two organization flows. Both Π_1 and Π_2 have five locations (labeled 'R', 'P', ..., 'D') where R, F are initial and A, D are final. Π_1 has enactments RPSM of length 4, RPSMPS of length 6, and RPSMPSMA of length 8 that is also complete. The organization flow Π_2 in Fig. 1(b) allows enactments FI(BI)*CC*D and their prefixes.

Note that the condition in the definition of enactment states that if an enactment reaches a terminal node, it should end (all but the last location should not be terminal).

Let $\Pi=(V, E, s, F)$ be an organization flow. For each integer $n\geqslant 0$, we define $E_\Pi(n)$ to be the set of all enactments of length n over Π, and $Ecnt_\Pi(n)$ as the cardinality of the set $E_\Pi(n)$, i.e., the number of distinct enactments over Π of length n.

Definition. Let Π be an organization flow. The *degree of nondeterminism of Π* is defined as $DN(\Pi) = max\{Ecnt_\Pi(n) \mid n \geqslant 0\}$. The degree of nondeterminism of Π is *bounded* if $DN(\Pi)$ is finite, *unbounded* otherwise.

For the organization flow Π_1 of Fig. 1(a), there are 2 enactments of length 1, i.e., $E_{\Pi_1}(1)=\{$RP, RA$\}$; also, $E_{\Pi_1}(2)=\{$RPA, RPS$\}$, $E_{\Pi_1}(3)=\{$RPSM$\}$, $E_{\Pi_1}(4)=\{$RPSMP, RPSMA$\}$, $E_{\Pi_1}(5)=\{$RPSMPS, RPSMPA$\}$, and $E_{\Pi_1}(6)=\{$RPSMPSM$\}$. In general, for each $n \geqslant 3$, $Ecnt_{\Pi_1}(n)=1$ if $n = 3k$ for some integer k, and $Ecnt_{\Pi_1}(n)=2$ otherwise. Therefore, $DN(\Pi_1)$ is bounded and is equal to 2.

For Π_2 in Fig.1(b), $E_{\Pi_2}(1)$={FI}, $E_{\Pi_2}(2)$={FIB, FIC}, $E_{\Pi_2}(3)$ = {FIBI, FICC, FICD}, $E_{\Pi_2}(4)$ = {FIBIB, FIBIC, FICCC, FICCD}, $E_{\Pi_2}(5)$ = {FIBIBI, FIBICC, FIBICD, FICCCC, FICCCD}, and so on. It is not too hard to see that $Ecnt_{\Pi_2}(n) = n$ and thus $DN(\Pi_2)$ is unbounded.

The following is the technical problem we study in this section: For a given organization flow Π, is $DN(\Pi)$ bounded? If it is, compute $DN(\Pi)$.

To decide the boundedness, one idea could be to first identify strongly connected components (SCCs) of an organization flow and then check if one SCC can reach another SCC. Although the existence of such SCCs correctly indicates unbounded degrees, the converse is not true since one SCC may already have unbounded degree of nondeterminism. In the following, we state a key property that nicely corresponds to the unbounded degree property.

Definition. Let Π be an organization flow and $r = v_0v_1\cdots v_n$ a complete enactment of Π. The enactment r is *dual-cyclic* if $r = uvxyz$ for some location sequences u, v, x, y, z such that

1. v, y, and z are nonempty, and
2. for each $i \geqslant 1$ and each $j \geqslant 1$, uv^ixy^jz is a complete enactment.

Lemma 2.2. Let Π be an organization flow. $DN(\Pi)$ is unbounded iff there exists a dual-cyclic enactment of Π.

Proof. (Sketch) For the "If" direction, let $r = uvxyz$ be a dual-cyclic complete enactment in Π where v, y, z are nonempty and uv^ixy^jz are complete enactments in Π for all $i, j \geqslant 1$. Let $k > 0$ be an arbitrary integer. Clearly $Ecnt_{\Pi}(n) > k$, if we choose $n = k \cdot |v| \cdot |y| + |uxz|$ ($|w|$ denotes the length of w). And thus $DN(\Pi)$ is unbounded.

For the "Only if" direction, we can translate Π into an equivalent finite state machine M over the alphabet of locations in Π. We can show that if $DN(\Pi)$ is not bounded, then there is some union-free regular expression contained in M with at least two stars ($*$). It can then be shown that Π has a dual-cyclic complete enactment. ∎

Intuitively, Lemma 2.2 states that the boundedness of nondeterminism degree depends on the nonexistence of two cycles on the way (enactment) to a terminal node in the graph of the organization flow. The first cycle would keep one active state and "send out" (through nondeterministic transitions) another active state to the second cycle. The second cycle does a similar thing for each active state it gets. Thus the number of enactments will keep increasing in the second cycle.

Theorem 2.3. Let $\Pi = (V, E, s, F)$ be an organization flow. It can be decided in $O(|V|+|E|)$ time if $DN(\Pi)$ is bounded.

We prove this result by constructing an algorithm that decides boundedness. Let Π be a fixed organization flow and treated as a graph. The algorithm consists of the following three main steps:

1. Identify nodes that lie on at least one complete enactment (path) of Π,
2. Identify cycles in Π, and
3. Identify connectivity between distinct cycles.

In Step 1, we remove nodes that are not on any complete enactment. This can be done by two runs of depth-first-search (DFS). If there are more than one terminal nodes, we pick one terminal node and connect all other terminal nodes to it. The first DFS starts from the (chosen) terminal node on the graph (with added edges) with all edges reversed their directions. It marks the nodes that are reachable, (i.e., there are paths from them to the terminal node). The second DFS starts from the initial node in the usual way and marks nodes reachable from initial location. All nodes missing a mark are removed.

During Step 2, we identify cycles (on some complete enactments). This step can be done by detecting *back edges* in the second DFS and marking all nodes which lie between the starting and end nodes of back edges.

During Step 3, we determine if there is a path between any two cycles. Since all cycles were identified during the second step, this step is reduced to searching paths between nodes that belong to two distinct cycles.

All three steps can be done during two DFS passes. The first DFS simply marks the nodes from which it can reach the final node. The second DFS is more intricate and accomplishes all three steps by traversing only the nodes marked during the first DFS.

We provide more detail of Steps 2 and 3 in the following. To detect back edges during the second DFS, for every node v, we use two types of vertex numbering:

1. *preorder*$[v]$: pre-order traverse numbering, and
2. *postorder*$[v]$: post-order traverse numbering

We also introduce a node marking *loop*$[v] =$ 'S', 'E', 'L', or '$\varnothing$' for each node $v \in V$. It encodes the order in which the cycle nodes are discovered by the DFS and distinguishes them from the other nodes not on any cycle. Initially, *loop*$[v]$='$\varnothing$' for every node v. As soon as a back edge (u, v) is detected by the DFS (starting from the initial node), locations u and v are marked as follows: *loop*$[u]$='E' and *loop*$[v]$='S'. Such markings define the boundary of the cycle, and allow us to mark all intermediate cycle members between u and v by 'L', when the DFS returns from its recursive calls.

We now focus on Step 3. First, we consider cycles sharing common nodes. Let *preorder*$[v]$ denote the DFS pre-order number of a node v. We define a single *global least preorder* number with the following operations:

- *reset GLP* : let *GLP* $= -1$ (when no cycles are being detected),
- *revise GLP* if (u, v) is a new-found back edge:
  (Case 1) set *GLP* $=$ *preorder*$[v]$ if current value *GLP* $= -1$,
  (Case 2) set *GLP* $=$ *min*(*preorder*$[v]$, *GLP*) if current value *GLP* $\neq -1$.

Intuitively, *GLP* is the smallest DFS preorder number of nodes on some cycle. It is easy to see that if *GLP*$(v) \neq -1$, then location v is inside some cycle. More specifically, when Case 1 of revising *GLP* happens, v is on a cycle. When Case 2 happens, then the parent of v belongs to a cycle, and one more back edge is detected. Hence, the node v is on two cycles. We call Case 2 *GLP double revision*.

We now consider detecting positive length paths between cycles. Since cycles are already identified, we only need to find nodes on a path from the initial node to a terminal node passing through nodes belonging to different cycles. To do this, we introduce a note marking *Path*$[v] = \{$*true*, or *false*$\}$ to indicate if a node v has a path to some cycle. The marking is assigned as follows, for every $v \in V$:

Algorithm 1. Deciding Boundedness

1. Set *GLP* $:= -1$; {Global Least Preorder number}
2. Set *loop*$[v] := \varnothing$ and *Path*$[v]$:= *false* for every $v \in V$;
3. *bounded* := *true*;
4. Let *Children*(u) be a set of children of node u;
5. Mark all nodes which lie on complete enactments with label '*R*';
6. *Boundedness*(s); {s is the initial node}

Algorithm 2. *Boundedness*(u)

```
1.  if u is unvisited, marked by 'R', and not in F then
2.    mark u as visited;
3.    for all v in Children(u), v ∉ F and marked with 'R' do
4.      if v is not visited then
5.        Boundedness(v);
6.      end if
7.      if Path[v] = true or (loop[v] ≠ 0 and postorder[v] ≠ 0 and GLP = −1) then
8.        Path[u] := true;
9.      end if
10.     if backedge detected then
11.       revise GLP;
12.       loop[u] := E; loop[v] := S;      {mark u and v as loop boundary nodes}
13.     end if
14.     if loop[u] ≠ S and loop[u] ≠ E and u ≠ v and GLP = −1 then
15.       loop[u] := L;      {mark u during back traverse as a loop node}
16.     end if
17.   end for
18. end if
19. if GLP has double revision or (loop[u] ≠ '∅' and Path[u] = true) then
20.   bounded := false;
21. end if
22. reset GLP;
```

1. *Path*$[v]$ = *false* if *postorder*$[v] = 0$ (DFS has not yet completed traversal on this node) or v does not have a path to any cycle.
2. *Path*$[v]$ = *true* if node has a path to some cycle.

If the organizational flow Π contains a node on a path from the initial node to a final node marked as ('S', *true*), ('L', *true*), or ('E', *true*), then there exist a path from the initial node to a terminal node passing through two cycles.

Algorithm 1 decides boundedness by setting the Boolean variable *bounded*. It initializes node markings and global variables *GLP* and *bounded* (Lines 1-3). Line 5 marks nodes having a path to a terminal node. In Line 6, procedure *Boundedness* is called on the initial node s, during which the variable *bounded* is changed to *false* if Π has an unbounded *DN*.

Algorithm 2 (*Boundedness*) processes only nodes that can reach a terminal node. *GLP* revision is in Line 11. The cycle detection is done in Lines 12, 14-15, detection of paths between cycles in Lines 7-8. At Line 20, the global flag *bounded* is set to *false* if path between cycles was detected (i.e. organization workflow has unbounded *DN*).

The complexity of *Boundedness* is the same as the time complexity of DFS. Hence, the total time complexity is $O(|V|+|E|)$.

We now consider the computation of the degree of nondeterminism of organization flows. The basic idea of the algorithm is to "simulate" the organization flow Π. The simulation is processed in topological order (all nodes in a cycle are collapsed into one). For every node and every execution length we store the numbers of paths arriving in this node. If a cycle is reached, cycle nodes will be propagated before proceeding to other nodes. If Π has a bounded *DN* then the total number of paths will be obtained at the end. The algorithm has two main steps:

1. Given a number $n \in N$ and a location $v \in V$, compute the number a_i of enactments of length i that ends in v (for every $i \in [1..n]$),
2. Find the maximum sum of a_i's of every $v \in V$ and every $i \in [1..n]$: $a_{\max} = max_{i\in[1..n]} \Sigma_{v\in V}(a_i)$.

In the first step, for each node $v \in V$ we keep a sequence of numbers such that position i in this sequence corresponds to the number of enactments of length i and ends in v. In the second step, for every node $v \in V$ we sum up all of sequences element by element. In the obtained sequence we find the maximum element.

We now discuss details of the first step. For every $v \in V$ we define $log(v) = (a_1, ..., a_i, ...)$ where a_i is the total number of enactments of length i that ends at node v.

For Fig. 1(a), $log(R)$ is $(1, 0, 0, ...)$. It means that there exists only one enactment of 1 ending in *R*. The $log(P)$ is $(0, 1, 0, 0, 1, 0, 0, 1, ...)$: there exists one enactment ending in *P* of lengths 2, 5, 8, and so on.

In the following discussion, we assume that if Π has bounded *DN*. For every $v \in V$, the length of $log(\cdot)$ equals to the maximum distance between the initial node and a terminal node, denoted as *MaxDist*. It can be shown that for organization flows with at most one cycle, the maximal value of a_i will appear in the first *MaxDist* elements of *log* for every node $v \in V$.

Lemma 2.4. If (V, E, s, F) is an organization flow with at most one cycle and $v \in V$ a location on a complete enactment, $log(v)$ can be computed in at most *MaxDist* steps.

Let Π be an organization flow such as $DN(\Pi)$ is bounded. For every node $v \in V$ the $log(v)$ of size no more than $Length(\Pi)$ is needed to compute *DN*.

We will compute enactments inside a loop and outside a loop separately. Consider two types of enactments:

1. *simple* enactments $v_1, ..., v_n$ where all v_i's are distinct, and
2. all other enactments (i.e., with repeated nodes) are called *composite*.

For every location v we split $log(v)$ into 2 parts:

1. $base[v] = (a^b_1, ..., a^b_{MaxDist})$, where a^b_i is the total number of simple and composite enactments of length i ending in node v and are directly from outside of the cycle.
2. $shift[v] = (a^s_1, ..., a^s_{MaxDist})$, where a^s_i is the total number of composite enactments of length i ending in node v and are contributed from inside the cycle.

Then, $log(v) = base[v] + shift[v]$.

To compute $base[\cdot]$ and $shift[\cdot]$ we need to know the period (size) of the cycle. Using node markings by procedure *Boundedness*, the period can be easily computed by modifying slightly the DFS part of the *Boundedness* procedure. To compute $log(v)$, we process nodes in the topological order. For the initial node s, $base[s] = (1, 0, 0, ...)$.

Algorithm 3. Compute Degree of Nondeterminism

1. Set *base*$[s] := (1, 0, ..., 0)$, and *base*$[v] := 0$ for every $v \in V - \{s\}$; $\{s$ is the initial node$\}$
2. Set *shift*$[v] := 0$ for every $v \in V$;
3. **for all** u in topological sort order **do**
4. **if** *loop*$[u]$ is not '$\varnothing$' {the first node in a cycle} **then**
5. **for all** nodes v on the cycle, starting from u **do**
6. Propogate *base*$[v]$ with period of the cycle;
7. **if** *base*$[v]$ is not a zero vector **then**
8. *shift* := RIGHT-SHIFT(*base*);
9. **for all** traversed nodes l starting from v in cycle **do**
10. *shift*$[l]$:= *shift*$[l]$ + *shift*;
11. *shift* := RIGHT-SHIFT(*shift*);
12. **end for**
13. **end if**
14. **end for**
15. **end if**
16. *base*$[u]$:= *base*$[u]$ + *shift*$[u]$;
17. **for** every v in *Children*$[u]$ **do**
18. **if** *loop*$[v]$ = '$\varnothing$' or *loop*$[u]$ = '$\varnothing$' **then**
19. *base*$[v]$:= *base*$[v]$ + RIGHT-SHIFT(*base*$[u]$);
20. **end if**
21. **end for**
22. **end for**
23. *DN*(Π) := $max_{a_i} \sum_{v \in V}$(*base*(v)); return *DN*(Π);

Now, for every $v \in V$ (in topological order) we do the following:

1. if *loop*$[v]$ = '$\varnothing$' then compute *base*$[v]$,
2. if *loop*$[v] \neq$ '$\varnothing$' then
   (a) if *loop*$[u]$ = '$\varnothing$' where u is a parent of v, then compute *base*$[u]$ in the same way as in step 1.
   (b) for every node l in the cycle, propagate *base*$[l]$ with its period P.
   (c) for every node l in cycle such that *base*$[l]$ is not equal to zero vector, traverse cycle once and compute *shift*$[l]$.

Let $\bar{v} = (n_1, ..., n_m)$ be a vector. We define the operation RIGHT-SHIFT$(\bar{v}) = (0, n_1, ..., n_{m-1})$. If a node u can reach node v, and *base*$[u] = (a_1^b, ..., a_{MaxDist}^b)$. The base of v is computed as follows (Step 1):

$$base[v] := base[v] + \text{RIGHT-SHIFT}(base[u]).$$

For the second step, we will first show how to propagate the base of the cycle members. If node v belongs to the cycle, then every a_i in *base*$[u]$ will appear with the frequency of the period P of the cycle. We will call such a log a *propagated base*$[v]$ *with period* P.

We also need to compute shift during cycle traversing. Let v be the cycle member such that *loop*$[v] \neq$ '$\varnothing$' and *base*$[v] \neq 0$, and let the cycle consist of the nodes $(v, v_1, v_2, ..., v_n)$. The shift for every node is defined recursively:

- *shift*$[v_1]$:= *shift*$[v_1]$ + RIGHT-SHIFT(*base*$[v]$).
- *shift*$[v_i]$:= *shift*$[v_i]$ + RIGHT-SHIFT(*shift*$[v_{i-1}]$), for each $i = 2, ..., n$.

Consider the organization flow Π_1 of Fig.1(a). *base*[R] $= (1, 0, 0, 0, 0)$. Node P is a loop member of cycle with period $= 3$. To obtain the *base*[P] we need to:

1. Shift the *base*[R]: *base*[P] $= (0, 1, 0, 0, 0)$,
2. Propagate log with period $P = 3$: *base*[P] $= (0, 1, 0, 0, 1)$

To obtain *shift*[S], we need to shift *base*[P], and so on.

After processing all nodes in topological order, we compute the degree of nondeterminism: for each node v, $log(v)$=*base*$[v]$+*shift*$[v]$ and $DN(\Pi) = max_i \sum_{v \in V}(log(v))$.

For the organization flow in Fig.1(a), log(R)=$(1, 0, 0, 0, 0)$, log(P)=$(0, 1, 0, 0, 1)$, log(S)=$(0, 0, 1, 0, 0)$, log(M)=$(0, 0, 0, 1, 0)$, log(A)=$(0, 1, 1, 0, 1)$. $DN(\Pi) = 2$.

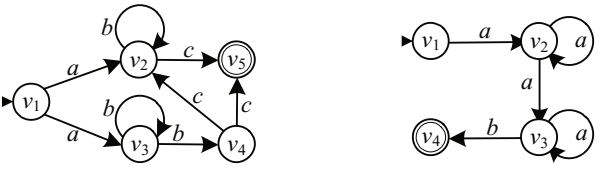
During topological order traversing, we compute base for children of every node, hence the complexity of this operation is $O(|V|+|E|)$. To traverse cycle, for every cycle member, we need to find the a child, with non-zero loop marking. Hence, the complexity of traversing is $O(|V|+|E|)$. Base propagation operation should be done for every non-zero element of base, hence the complexity is $O(V)$. The total complexity of the algorithm is at most $O(|V|(|V|+|E|)^2)$. However, each edge will then be traversed at most $|V|$ (one for each node on a cycle) times. Since the size of *log* vectors is at most $|V|$, the total complexity of the algorithm is $O(|V|^2|E|)$. Note that the algorithm works correctly if the input organization flow has at most one cycle (and thus bounded). For the general case, the complexity of computing the degree is higher.

3 Workflows and Executions

In this section we present a general model for workflows that are essentially compositions of services using finite state machines (FSAs). The FSA part of the workflow model is roughly the Roman service model [2,5,10]. The formal model is extended from the organization flow model in Section 2 in two aspects. First, edges are labeled with task names to reflect the activities to be done along the edge. Second, locations are indirectly modeled by mappings that assign states in FSAs to locations. This allows the ability to model tasks and locations similar to modeling workflows with "lanes" in BPMN. Due to these extensions, the notion of an enactment includes task names and the notion of degree of nondeterminism also considers "co-location" of enactments.

We now outline the general model and revise the core notions including degree of nondeterminism below. Intuitively, at the core of a workflow is a FSA. The two main differences of the notion of nondeterminism degree for workflows and for organization flows are (1) the former will take into consideration of input word (or task sequence), and (2) if a proper prefix w' of a word w is accepted, the execution of the portion of w beyond w' will not happen since the execution halts on the first opportunity. We note that in an organization flow, if there is a complete enactment of length ℓ, there may still be enactments of some lengths $> \ell$ that still contribute to the degree of nondeterminism. It is interesting to note that if we only make the first change, the resulting notion would measure the "degree of parallelism", which is not studied in this paper.

Example 3.1. Fig. 2(a) shows a travel planning workflow for a travel agency with nodes denotes the desks (agents) and edge labels denote tasks to be performed. In the example, a denotes getting request details, b represents revising the draft itinerary (possibly



(a) Travel planning (Π_3) (b) Order fulfillment (Π_4)

Fig. 2. Example Workflows

interacting with the customer), c is for processing payments. A customer could make a supplemental for additional changes made after the first payment. The agents could be in the same room in which their verbal communication could help managing the service for a customer. The location information isn't specified in the figure. An order fulfillment workflow is shown in Fig. 2(b) with two warehouses (v_2, v_3). ∎

As illustrated in Example 3.1, a node in a workflow may represent the control point of a task or service, or a service provider (i.e., a location or lane), and should have the execution flow control during the execution; an edge typically represents a request (e.g., a requested task) or transmission of the control.

In the formal setting, we use standard nondeterministic finite state machines. We briefly review the key (standard) concepts, and omit some details (see [8]).

A *(nondeterministic) finite state machine* (*FSA*) is a tuple $(\Sigma, V, s, F, \Delta)$ where Σ is a finite set (i.e. *alphabet*) of *symbols*, V a finite set of *states*, $s \in V$ the *initial* state, $F \subseteq V$ a set of *final* (or *terminal*) states, and $\Delta \subseteq V \times \Sigma \times V$ a *transition* relation.

Let $n \geqslant 0$ be an integer, $M = (\Sigma, V, s, F, \Delta)$ an FSA, and $w = a_1 \cdots a_n$ a word of length n over Σ. A *run* (*of* w) (*in* M) is an alternating sequence of states and symbols $r = v_0 a_1 v_1 a_2 \cdots a_n v_n$ such that $v_0 = s$ (the initial state) and for each $i \in [1..n]$, $v_i \in V$, and $(v_{i-1}, a_i, v_i) \in \Delta$ is a transition in M. Furthermore, the run r is *terminating* if $v_n \in F$. The word w is *accepted by* M if there is a terminating run of w in M. Let $\mathcal{L}(M)$ denote set of all words accepted by M and $\textsc{run}_M(w)$ denote the set of all runs of w over M. We denote $\textsc{run}_M(w)$ simply as $\textsc{run}(w)$ if M is clear from the context.

We now define workflows, which FSAs with the following modifications:

1. Each state is assigned a location indicating where the execution (control point) is currently located, and
2. Each symbol in the alphabet of the FSA is viewed as an activity or "task".

The treatment of symbols as tasks (Item 2) resembles the Roman model of services [2,5]; in the remainder of this paper, we will use the words "symbol" and "task" interchangeably. The inclusion of locations seems new in formal models, though practically it is hardly new. Intuitively, a location corresponds to the concept of a "lane" in BPMN.

Definition. A *workflow (schema)* is a pair $\Pi = (M, \lambda)$ where M is an FSA with a set V of states, and λ is a total mapping from V into $\boldsymbol{L}$ (locations).

Note that different states can be mapped to the same location. For convenience, we may also refer to states as "nodes".

Since the core of a workflow $\Pi = (M, \lambda)$ is an FSA M, the notions (e.g., runs) are easily carried over with an important change: a workflow "enactment" requires to be "prefix-free", as explained below. Intuitively, no enactments should have a proper prefix w as a complete execution (enactment). Note that this is defined semantically (rather than syntactically).

A word w' is a *(proper) prefix* of another w if there is a (resp. nonempty) word u such that $w = w'u$. A set of words is *prefix-free* if it does not contain two words such that one is a proper prefix of another.

Definition. Let $\Pi = (M, \lambda)$ be a workflow over a set Σ of tasks. A word w over Σ is a *request in* Π if every proper prefix of w is not in $\mathcal{L}(M)$. A request is *complete* if it is in $\mathcal{L}(M)$. The *capability* of Π, denoted as $\text{CP}(\Pi)$, is a set of all complete requests in Π.

Alternatively, the capability of a workflow is the maximum prefix-free subset of the language accepted by the FSA in the workflow. Complete workflow requests are supported by (executions of) the workflow.

Consider the workflow Π_3 in Fig. 2(a) with five states and three tasks. The word $abbc$ is a complete request, abb is a request but not complete. The word $abbcc$ is not a request since its proper prefix $abbc$ is a complete request.

We now extend the central notion of degree of nondeterminism. Consider a workflow Π with its nodes spread over different locations. If we are to execute Π for a request w, we need a mechanism for controlling the execution flow. If the FSA in Π is deterministic, the flow control management is simple, e.g., once the execution completes at the current location (state), the control moves to the the unique next location (state). However, the management is complicated when Π is nondeterministic. For example, the current location (state) v may have two possible next locations (states) v' and v''. In this case, the execution control at v will split to ***two*** execution control points at v and v', resp. Both need to be managed for further execution of w. It is conceivable that such split could happen throughout the execution for a workflow request. In the most general case, there may not even be an upper bound on the number of control points at one time instant. The remainder of this paper focuses on the maximum number of such execution control points to be managed to complete executions for workflow requests.

Let $\Pi = (M, \lambda)$ be a workflow where $M = (\Sigma, V, s, F, \Delta)$. We extend the location mapping λ to the domain $V \cup \Sigma$ such that $\lambda(a) = \epsilon$ (the empty word) for each $a \in \Sigma$.

Definition. Let $\Pi = (M, \lambda)$ be a workflow and w a request in Π. Each run in $\text{RUN}(w)$ is also an *enactment* of Π. Two enactments π, π' of w are *co-located w.r.t.* w if $\lambda(\pi) = \lambda(\pi')$, i.e., they travel through the same sequence of locations.

Assuming Π, M the same as above, it is easy to see that co-location is an equivalence relation over the set $\text{RUN}(w)$ for each workflow request w. For each workflow request w, we define the set $\text{EXEC}(w) = \{[r] \mid r \in \text{RUN}(w)\}$, where $[r]$ denotes the equivalence class containing r. Furthermore, we define $\mathit{Ecnt}(w)$ to be the cardinality of $\text{EXEC}(w)$.

Definition. The *degree of nondeterminism* of a workflow Π is defined as: $\mathit{DN}(\Pi) = \mathit{max}\{\mathit{Ecnt}(w) \mid w$ is a proper prefix of $u, u \in \text{CP}(\Pi)\}$. The degree of nondeterminism of Π is *bounded* if $\mathit{DN}(\Pi)$ is finite, *unbounded* otherwise.

Example 3.2. For workflow Π_3 in Fig.2(a), assuming states are assigned distinct locations,

$\text{RUN}(a) = \{\ell_1 a \ell_2, \ell_1 a \ell_3\}$ and $\mathit{Ecnt}(a) = 2$,
$\text{RUN}(ab) = \{\ell_1 a \ell_2 b \ell_2, \ell_1 a \ell_3 b \ell_3, \ell_1 a \ell_3 b \ell_4\}$ and $\mathit{Ecnt}(ab) = 3$,
$\text{RUN}(abb) = \{\ell_1 a \ell_2 b \ell_2 b \ell_2, \ell_1 a \ell_3 b \ell_3 b \ell_3, \ell_1 a \ell_3 b \ell_3 b \ell_4\}$ and $\mathit{Ecnt}(abb) = 3$,

$\mathit{Ecnt}(abc) = 3$, and $\mathit{Ecnt}(abbc) = 3$. It can be shown that $\mathit{DN}(\Pi_3)$ is bounded and $= 3$. It is interesting to note that if we view Π_3 as an organization flow (i.e., ignore the edge labels), the degree of nondeterminism is actually unbounded. For workflow Π_4 in Fig.2(b) with states assigned distinct locations,

$\text{RUN}(a) = \{\ell_1 a \ell_2\}$,
$\text{RUN}(aa) = \{\ell_1 a \ell_2 a \ell_2, \ell_1 a \ell_2 a \ell_3\}$,
$\text{RUN}(aaa) = \{\ell_1 a \ell_2 a \ell_2 a \ell_2, \ell_1 a \ell_2 a \ell_2 a \ell_3, \ell_1 a \ell_2 a \ell_3 a \ell_3\}$,

and so on. It is not too hard to see that $\mathit{Ecnt}(a^n) = n$ and thus $\mathit{DN}(\Pi_4)$ is unbounded. ∎

4 Management of Workflow Executions

In this section, we focus on management of workflow executions. In particular, we study the problems of deciding if the degree of nondeterminism is bounded and computing the bound. We consider two subclasses of workflows, "fully distributed" workflows where each state is assigned a distinct location, and "transformation flows" where all states are assigned the same location. For the former, we show that the problems are solvable and outline the algorithms. For the latter, the degree of nondeterminism is always 1 by definition. However, one can still use the same techniques to minimize the number of enactments needed during execution. We show that each transformation flow can be optimally executed with only 1 enactment.

4.1 Fully Distributed Workflows

We now consider the first subclass of workflows called fully distributed workflows.

Definition. A workflow $\Pi = (M, \lambda)$ is *fully distributed* if λ is a one-to-one mapping.

Theorem 4.1. It can be decided if a given fully distributed workflow has a bounded degree of nondeterminism.

We now briefly discuss the proof of the above theorem. The key idea is to reduce the problem to the boundedness problem of "vector addition" systems [14].

A *vector addition system* is a triple $S = (k, z_0, A)$ where k is a non-negative integer, z_0 a k-vector of integers, and A a finite set of k-vectors of integers. A *path to a vector* z in S is a sequence $z_0, z_1, ..., z_n$ such that $z = z_n$ and for each $i \in [1..n]$, $z_i - z_{i-1} \in A$. A vector addition system is bounded if there exists an integer ℓ such that whenever there is a path to z, $z.i$ (the i-th component of z) is $\leqslant \ell$.

We now proceed to describe the reduction. Let $\Pi = (M, \lambda)$ be a fully distributed workflow where $M = (\Sigma, V, s, F, \Delta)$ is an FSA. We fix Π and M in the following discussions.

Lemma 4.2. For each request w of Π and two enactments r, r' in $\text{RUN}(w)$ are co-located iff $r = r'$ (they are identical).

The above lemma follows from the fact that the workflow Π is fully distributed.

The semantics of workflow restricts workflow requests to only those words over Σ that have no proper prefixes accepted by the FSA M. Thus not every run in M is an enactment. Thus we need the following.

Lemma 4.3. $\text{CP}(\Pi)$ is a regular language.

It is easy to see that $\text{CP}(\Pi)$ is the maximal prefix-free subset of $\mathcal{L}(M)$. Results from [7,18] states that such maximal prefix-free subsets of regular languages are also regular. Let M' be an FSA such that $\mathcal{L}(M') = \text{CP}(\Pi)$.

We now construct the (standard) product FSA M'' of M and M' such that $\mathcal{L}(M'') = \mathcal{L}(M) \cap \mathcal{L}(M') = \mathcal{L}(M) \cap \text{CP}(\Pi)$. States in M'' are pairs of states from M and M' respectively. Without loss of generality, we further assume that each state in M'' is reachable from the initial state and can reach a final state. We extend the mapping λ to a mapping λ' over states in M'' such that it assigns locations according to λ.

Lemma 4.4. The following are all true.

1. $\Pi' = (M'', \lambda')$ is a workflow.
2. $\text{CP}(\Pi') = \text{CP}(\Pi)$.
3. For each prefix w of each word in $\mathcal{L}(M'')$, there is a 1-1 mapping from $\text{RUN}_{\Pi'}(w)$ into $\text{RUN}_{\Pi}(w)$.
4. $DN(\Pi') = DN(\Pi)$.

We now describe the reduction from Π' to vector addition systems. Let k be the number of states in M''. Without loss of generality, let $v_1, ..., v_k$ be an enumeration of states in M'' and v_1 is the initial state. Let $z_0 = (1, 0, ..., 0)$. We construct the set A of vectors according to transitions. If on state v_3 and symbol a, the next possible states are $\{v_2, v_4\}$, we construct a vector $(0, 1, -1, 1, 0, ..., 0)$. It is straightforward to see that the vector addition system is bounded iff $DN(\Pi')$ is finite. The decidability of boundedness for degree of nondeterminism follows from [14], with an exponential space complexity.

Theorem 4.5. Let Π be a fully distributed workflow such that $DN(\Pi)$ exists (i.e., bounded). Then, $DN(\Pi)$ can be effectively computed.

The proof of Theorem 4.5 is very similar to the proof of Theorem 4.1. Specifically, we also construct the FSA M'' that captured the prefix-free set $\text{CP}(\Pi)$ and the workflow Π'. Instead of mapping Π' to a vector addition system, we construct a Petri net [13] using the idea similar. Specifically, each state is mapped to a place, a set of transition from a state on an input symbol is mapped to a transition. Using the technique in [4], we can construct a "coverability graph" of the Petri net, from which we can find the largest number of enactments at one time instant, i.e., $DN(\Pi')$. By Lemma 4.4, $DN(\Pi)$ is obtained. Since the coverability graph is exponential to the input Petri net, it follows that the complexity of this problem is also in exponential space.

4.2 Transformation Flow

In this section we focus on a subclass of workflows called "transformation flows". Intuitively, transformation flows are insensitive to locations; specifically, all states are

mapped to a predetermined location. In this case, the degree of nondeterminism is always 1, according to the definition. This means that nondeterministic executions can be managed since they are all contained in one location.

However, for each request w in a workflow Π, the set $\text{RUN}_{\Pi}(w)$ of enactments for the request still needs to be managed during the execution. In this subsection, we use the standard techniques to show that we could construct an alternative workflow Π' such that Π and Π' have the same capability (i.e., are "equivalent") and for each request in Π', there is a unique enactment.

We now proceed with the technical discussions.

Definition. A workflow (M, λ) is *transformational* if λ is a contant function.

Theorem 4.6. For each transformational flow Π, we can effectively construct another transformational flow Π' such that $\text{CP}(\Pi) = \text{CP}(\Pi')$ and for each request w, $\text{RUN}_{\Pi'}(w)$ is a singleton set.

Let $\Pi = (M, \lambda)$ be a transformational flow. The key idea in optimizing execution of a nondeterministic transformation flow is to construct an equivalent deterministic FSA. By Lemma 4.3, $\text{CP}(\Pi)$ is a regular language. Therefore, there exists a deterministic FSA M' such that $\mathcal{L}(M'') = \text{CP}(\Pi)$. Letting $\Pi' = (M', \lambda)$, it is easy to see that Π' satisfies the conditions in Theorem 4.6. Finally, we note that the key construction here is to turn a nondeterministic FSA into a deterministic one; combining results from [7,18] on constructing prefix-free subset, the complexity of constructing Π' is thus exponential time.

5 Conclusions

In this paper we formulate a new notion of degree of nondeterminism for nondeterminism workflows and service compositions and initiate a study on the decision and computation problems. Although preliminary results are obtained, many interesting questions remain to be explored. For example, can Algorithm 3 be generalized and improved? Also, it is not clear if the complexity upper bounds of the decision and computation problems for fully distributed workflows can be further improved.

References

1. Berardi, D., Calvanese, D., De Giacomo, G., Mecella, M.: Composition of services with nondeterministic observable behavior. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 520–526. Springer, Heidelberg (2005)
2. Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 43–58. Springer, Heidelberg (2003)
3. Berardi, D., De Giacomo, G., Mecella, M., Calvanese, D.: Composing web services with nondeterministic behavior. In: Proc. IEEE Int. Conf. Web Services, ICWS (2006)
4. Finkel, A.: The minimal coverability graph for petri nets. In: Proc. 12th Int. Conf. on Applications and Theory of Petri Nets: Advances in Petri Nets, pp. 210–243 (1991)

5. Gerede, C., Hull, R., Ibarra, O., Su, J.: Automated composition of e-services: Lookaheads. In: Proc. 2nd Int. Conf. on Service-Oriented Computing, ICSOC (2004)
6. Hackmann, G., Haitjema, M., Gill, C., Roman, G.-C.: Silver: A BPEL workflow process execution engine for mobile devices. In: Proc. Int. Conf. Service Oriented Computing, ICSOC (2006)
7. Han, Y.-S., Wang, Y., Wood, D.: Prefix-free regular languages and pattern matching. Theoretical Computer Science 389, 307–317 (2007)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
9. Hull, R., Benedikt, M., Christophides, V., Su, J.: E-services: A look behind the curtain. In: Proc. ACM Symp. on Principles of Database Systems (2003)
10. Hull, R., Su, J.: Tools for composite web services: A short overview. SIGMOD Record 34(2), 86–95 (2005)
11. Lemcke, J., Friesen, A.: Composing web-service-like abstract state machines (asms). In: Proc. IEEE Congress on Services, SERVICES (2007)
12. Liu, G., Liu, X., Qin, H., Su, J., Yan, Z., Zhang, L.: Automated realization of business workflow specification. In: Proc. Int. Workshop on SOA, Globalization, People, and Work (2009)
13. Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: Proc. of IFIP Congress 1962, pp. 386–390. North Holland Publ. Comp., Amsterdam (1963)
14. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoretical Computer Science (1978)
15. Sakata, Y., Yokoyama, K., Matsuda, S.: A method for composing process of nondeterministic web services. In: Proc. IEEE Int. Conf. Web Services, ICWS (2004)
16. Su, J.: Special Issue on Semantic Web Services: Composition and Analysis. IEEE Data Eng. Bull. 31(3) (September 2008), http://sites.computer.org/debull/A08sept/issue1.htm
17. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
18. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Word, Language, Grammar. Handbook of Formal Languages, vol. 1, pp. 41–110. Springer, Heidelberg (1997)
19. Zeng, L., Flaxer, D., Chang, H., Jeng, J.-J.: PLM$_{flow}$-Dynamic business process composition and execution by rule inference. In: Proc. 3rd Int. Workshop on Technologies for E-Services (TES). Springer, Heidelberg (2002)
20. Zhang, X., Junqueira, F., Hiltunen, M.A., Marzullo, K., Schlichting, R.D.: Replicating nondeterministic services on grid environments. In: Proc. 15th IEEE Int. Symp. on High Performance Distributed Computing (2006)

Author Index