UNIVERSITY OF OSLO
Department of Informatics

FLACOS'09 Workshop Proceedings

Research Report No. 385

Gordon J. Pace

Gerardo Schneider

ISBN 82-7368-345-1
ISSN 0806-3036

September 2009

Proceedings of

FLACOS'09

Third Workshop on Formal Languages and Analysis of Contract-Oriented Software

24–25 September 2009

Toledo, Spain

Gordon J. Pace and Gerardo Schneider (editors)

Foreword

The 3rd Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'09) is held in Toledo, Spain. The aim of the workshop is to bring together researchers and practitioners working on language-based solutions to contract-oriented software development.
The workshop is partially funded by the Nordunet3 project "COSoDIS" (Contract-Oriented Software Development for Internet Services).

The program consists of 3 regular papers, 6 invited participant presentations, and presentation from the COSOoDIS team. The regular papers were selected by the following programme committee:

Björn Bjurling	SICS, Sweden
Olaf Owe	University of Oslo, Norway
Gordon J. Pace	University of Malta, Malta (co-chair)
Anders P. Ravn	Aalborg University, Denmark
Gerardo Schneider	Chalmers|Göteborg University, Sweden
	University of Oslo, Norway (co-chair)
Valentin Valero Ruiz	University of Castilla-La Mancha, Spain

The local organization was chaired by M. Emilia Cambronero Piqueras from the University of Castilla-La Mancha, together with the following team:

Enrique Arias Antúnez	University of Castilla-La Mancha, Spain
Antonio Bueno Aroca	University of Castilla-La Mancha, Spain
Diego Cazorla López	University of Castilla-La Mancha, Spain
Fernando Cuartero Gómez	University of Castilla-La Mancha, Spain
Gregorio Díaz Descalzo	University of Castilla-La Mancha, Spain
Fernando López Pelayo	University of Castilla-La Mancha, Spain
Hermenegilda Macià Soler	University of Castilla-La Mancha, Spain
Enrique Martínez López	University of Castilla-La Mancha, Spain
M. Carmen Ruiz Delgado	University of Castilla-La Mancha, Spain
Valentín Valero Ruiz	University of Castilla-La Mancha, Spain

Further information can be found at the workshop homepage: `http://www.ifi.uio.no/flacos09`. This website was developed and maintained by Enrique Martínez López.

Acknowledgments

Besides the Nordunet3 committee, we thank the Ministerio de Educación with the project TIN2006-15578-C02-02, the JCCM with the regional project PEII09-0232-7745, the Albacete Science & Technology Park, the I^{3}A (Computer Science Research Institute of Albacete), the ESII (School of Computer Science of Albacete) and the Computer System Department of the University of Castilla-La Mancha for their support.
We also thank the President of the Regional Courts of Castilla-La Mancha for welcoming us and contributing with an official reception.

Welcome to Toledo!
Gordon J. Pace and Gerardo Schneider

Table of Contents

Evolution of Contracts*

Gilles Barthe

IMDEA Software
Madrid, Spain
gilles.barthe@imdea.org

Abstract. Any formalism to describe contracts must be able to capture evolvability over time, and also to correlate such evolutions to changes in the environment or in the behavior of the parties involved in contracts. Yet, few works have focused on the general problem of verifying evolvable contracts.
The talk will show that verification techniques for static contracts naturally extend to evolvable contracts. Starting from a very general view of contracts as syntactic entities that characterize sets of traces, we show how to accomodate two essential ingredients of dynamic contracts: spillover, which characterizes the remains of a clause when it is withdrawn from a contract, and power, which characterizes when a principal is entitled to perform a change in a contract. Then, we show that standard definitions of offline and online monitoring extend to dynamic contracts, and we prove that the standard criteria of soundness and completeness of online monitoring wrt offline monitoring remain applicable for dynamic contracts. One useful consequence of our results is the possibility of relying on online methods to ensure that power is used correctly. Although the technical development is carried in an abstract setting, we illustrate our definitions and results using contract languages for rights and obligations; these languages, despite their simplicity, share many essential features with other formalisms for digital right management and access control, and are therefore representative of the potential interest of our approach.

* Joint work with Gordon Pace and Gerardo Schneider

On Behavioural Interfaces and Contracts for Software Adaptation

Javier Cámara, José Antonio Martín, Gwen Salaün, Carlos Canal, Ernesto Pimentel

Department of Computer Science, University of Málaga, Spain

Abstract

Software Adaptation aims at composing in a non-intrusive way black-box components or services, even if they present some mismatches in their interfaces. Adaptation is a complex issue especially when behavioural descriptions of services are taken into account in their interfaces. In this paper, we first present our abstract notations used to specify behavioural interfaces and adaptation contracts, and propose some solutions to support the specification of these contracts. Then, we overview our techniques for the generation of centralized or distributed adaptor protocols and code based on the aforementioned contracts.

1. Introduction

Service-based systems are built by reusing existing components and services. These services can be used to fulfill basic requirements, or be composed with other services to build bigger systems which aim at working out complex tasks. Services must be equipped with rich interfaces enabling external access to their functionality which can be described at different interoperability levels (*i.e.*, signature, protocol, quality of service, and semantics). Composition of services is seldom achieved seamlessly because mismatch may occur at the different interoperability levels and must be solved. *Software adaptation* is the only way to compose non-intrusively black-box components or services with mismatching interfaces by automatically generating mediating *adaptor* services. Adaptation goes beyond classic composition of components or services since in these approaches, see for instance [1, 2, 3], no solution is proposed to compensate possible differences existing between incompatible interfaces.

So far, most adaptation approaches have assumed interfaces described by signatures (operation names and types) and behaviours (interaction protocols). Describing protocol in service interfaces is essential because erroneous executions or deadlock situations may occur if the designer does not consider them while building composite services. Deriving adaptors is a complicated task since, in order to avoid undesirable behaviours, the different behavioural constraints of the composition must be respected, and the correct execution order of the messages exchanged must be preserved.

Most existing works on model-based behavioural adaptation (see for instance [4, 5, 6]) favour the full automation of the process. They are referred to as *restrictive approaches* because they try to solve interoperability issues by pruning the behaviours that may lead to mismatch, thus restricting the functionality of the services involved. These techniques are limited since they are not able to fix subtle incompatibilities between service protocols by remembering and reordering events and data when necessary. A second class of solution is referred to as *generative approaches* (see for instance [7, 8, 9]). These avoid restricting service behaviour, and support the specification of advanced adaptation scenarios. Generative approaches build adaptors automatically from an abstract specification, namely an *adaptation contract*, of how mismatch cases can be solved.

Manual writing of an adaptation contract is a difficult and error-prone task. In particular, incorrect correspondences between operations in service interfaces, or syntactic mistakes are common, especially when the contract has to be specified using cumbersome textual notations. Moreover, a contract is just an abstract specification of how the different services should interact and does not explicitly describe all the different execution scenarios of a system, which may not be easily envisioned by the designer. Finally, writing a contract requires a good comprehension of the services involved, and understanding all the details of service protocols is quite complicated for non-experts.

Email addresses: jcamara@lcc.uma.es (Javier Cámara), jamartin@lcc.uma.es (José Antonio Martín), salaun@lcc.uma.es (Gwen Salaün), canal@lcc.uma.es (Carlos Canal), ernesto@lcc.uma.es (Ernesto Pimentel)

In this paper, we present an approach that fully supports generative adaptation, which starts with the automatic extraction of behavioural models from existing interface descriptions either in Abstract BPEL or Windows Workflows (WF), and ends with the generation of a monolithic adaptor or a set of distributed adaptation wrappers that are automatically generated and deployed. We will present the different parts of our solution with a particular focus on the notations used here to specify behavioural interfaces and adaptation contracts. More precisely, we will present two alternatives to manual contract specification. A first one, namely *automatic contract specification*, aims at constructing adaptation contracts without any human intervention. A second one, namely *interactive contract specification*, supports the user through the adaptation contract design process using a graphical notation and interactively pointing out suggestions and inconsistencies in the design by using protocol similarity, simulation and verification techniques. We also propose a combined use of both approaches. Our approach is fully supported by a toolbox called ITACA.

The rest of this paper is structured as follows: Section 2 presents our service model and some techniques supporting the contract specification. In Section 3, we overview our solutions to generate the adaptor protocol and code from the behavioural interfaces and adaptation contract. Section 4 presents our tool support and Section 5 some concluding remarks.

2. Behavioural Interfaces and Adaptation Contracts

2.1. Behavioural Interfaces

We assume that service interfaces are specified using both a signature and a protocol. Signatures correspond to operation names associated with arguments and return types relative to the messages and data being exchanged when the operation is called. Protocols are represented by means of *Symbolic Transition Systems* (STSs), which are Labelled Transition Systems (LTSs) extended with value passing [10]. Communication between services is represented using events relative to the emission and reception of messages corresponding to operation calls. Events may come with a set of data terms whose types respect the operation signatures.

This formal model has been chosen because it is simple, graphical, and provides a good level of abstraction to tackle verification, composition, or adaptation issues [11, 12, 13]. At the user level, one can specify service interfaces (signatures and protocols) using respectively WSDL, and Abstract BPEL (ABPEL) or WF workflows (AWF) [14]. These, are automatically parsed and translated into our internal STS model.

2.2. Adaptation Contract Specification

An adaptation contract [8] contains an interface mapping matching operations (including their arguments) required by service interfaces with those offered by others in order to reconcile interface mismatch at the signature and behavioural levels. Furthermore, a contract may also contain additional properties to be imposed on the composition of the different services, such as specific orderings on operation invocations. Therefore, understanding how two protocols differ helps to build adaptation contracts, for instance by suggesting the best possible operation matches to the user. To do so, our approach is able to compute protocol similarities [15], which aim at pointing out differences between protocols, but also at detecting parts of them which turn out to be similar. Our similarity computation relies on a *divide-and-conquer* approach to compute the similarity of service protocols (described as STSs) from a set of detailed similarity comparisons (states, labels, depths and graphs). This information can be used to guide the contract specification process, regardless of the specific technique employed. In particular, we introduce in this section our contract notation and two different specification techniques for adaptation contracts:

Notation. Our adaptation language makes communication among services explicit, and specifies how to work out mismatch situations. To make communication explicit, we rely on *vectors* (inspired from synchronization vectors [16]), which denote communication between several services, where each event appearing in one vector is executed by one service and the overall result corresponds to an interaction between all the involved services. A vector may involve any number of services and does not require interactions to occur on the same names of events. Vectors express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. Furthermore, variables are used as placeholders in message parameters. The same variable name appearing in different labels (possibly in different vectors) enables the relation of sent and received arguments of messages.

In addition, the contract notation includes an LTS with vectors on transitions (vector LTS or VLTS). This is used as a guide in the application order of the interactions denoted by vectors. VLTSs go beyond port and parameter bindings, and

express more advanced adaptation properties (such as imposing a sequence of vectors or a choice between some of them). If the application order of vectors does not matter, the vector LTS contains a single state with all transitions looping on it.

Automatic Contract Specification. In order to alleviate the cumbersome task of designing adaptation contracts and to avoid mistakes in the specification (which may lead to undesirable behaviours of the system), we can use the above mentioned similarity measures for the automatic generation of contracts [17]. This automatic contract generation is achieved traversing the behaviour of the services and matching the different operations found based on similarity measures. In such a way, we are able to match compatible operations and to adapt the minimum set of operations required for the deadlock-free composition of services. The generated contracts successfully specify how to overcome signature mismatch (*i.e.*, different operation names and arguments) and behavioural incompatibilities (*i.e.*, message splitting/merging, missing messages and message reordering) in such a way that all services are able to interact with each other and reach a correct termination state of their execution.

Interactive Contract Specification. Automatic contract generation may produce solutions leading to deadlock-free compositions unable to fulfill their intended goals, since the automatic approach is not currently aware of the underlying semantics of the services. Therefore, our approach incorporates an Adaptation Contract Interactive Design Environment [18], which aims at helping the designer in specifying a contract, reducing the risk of errors introduced by manual specification. In contrast with using textual notations where the designer can write any (correct or incorrect) statement, our environment makes use of a graphical notation which enables interactive and incremental construction and checks on the contract. Thus, any contract produced with our proposal is syntactically correct and consistent. In addition, the interactive environment is able to:

- Assist the designer by pointing out the best matches between ports graphically using protocol similarity information.
- Simulate the execution of the system step-by-step and determine how the different behavioural interfaces evolve as the different parts of the contract are executed, highlighting active states and fired transitions on the graphical representation of interfaces.
- Automatically identify execution traces leading to deadlock or livelock. These can be replayed step-by-step using simulation to understand the cause of the incorrect behaviour. This helps the designer to detect the behavioural issues that might be raised during execution and to understand if the behaviour of the system complies with his/her design intentions.

It is worth observing that the automatic and interactive approaches mutually improve their results when they are combined. On one hand, when the automatic contract specification process receives adaptation constraints from the interactive design environment, it is able to discard solutions leading to deadlock-free compositions that may not fulfill their intended goals (*e*.g., a client-supplier system which always aborts requests). On the other hand, the designer can use the automatic approach to complete parts of a contract through the interactive environment.

3. Adaptor Generation and Implementation

From a set of service protocols and a contract specification, we can generate either an *adaptor* protocol (centralized view), or a set of *adaptation wrapper* protocols (distributed view). In the first case, the adaptor can be deployed on a single machine. In the case of wrappers, they can be distributed and deployed using middleware technologies, preserving a full parallelism of the system's execution. Adaptor and wrapper protocols are automatically generated in two steps: (i) system's constraints are encoded into the LOTOS [19] process algebra, and (ii) adaptor and wrapper protocols are computed from this encoding using on-the-fly exploration and reduction techniques. Beyond simulation and verification techniques integrated in the interactive environment, the LOTOS encoding allows to check temporal logic properties on the adaptor under construction using the CADP model-checker [20]. The reader interested in more details may refer to [10, 21].

Our internal model (STS) can take into account some additional behaviours (interleavings) that cannot be implemented into executable languages. To make platform-independent adaptor protocols (obtained in the former step) implementable *wrt.* a specific platform (*e.g.*, BPEL), we proceed in two steps: (i) filtering the interleaving cases that cannot be implemented (*e.g.*, several emissions and receptions outgoing from a same state), and (ii) encoding the filtered model into the corresponding implementation language. Following the guidelines presented in [10], the adaptor protocol is implemented as a BPEL process using a state machine pattern. The main body of the BPEL process corresponds to a global *while* activity with *if* statements

used inside it to encode adaptor states. Each *if* body encodes transitions outgoing from the corresponding state. Variables are used to store data passing through the adaptor and the current state of the protocol.

4. Tool Support

Our solution for model-based software adaptation overviewed in this paper is fully supported by ITACA [22], an integrated toolbox we implemented (see Fig. 1). ITACA has been implemented in Python and Java, and consists of about 51,000 lines of code. We have intensively applied and validated our toolbox on many case studies such as a travel agency, rate finder services, on-line computer material store, library management systems, SQL servers, and many other systems.

Although our toolbox automates all the steps of the adaptation process, contract specification requires human intervention to ensure that the goal of the composition is fulfilled. However, experiments we have carried out show that the techniques proposed in ITACA to support the adaptation contract construction drastically reduce the time spent to build the contract and the number of errors made during this process.

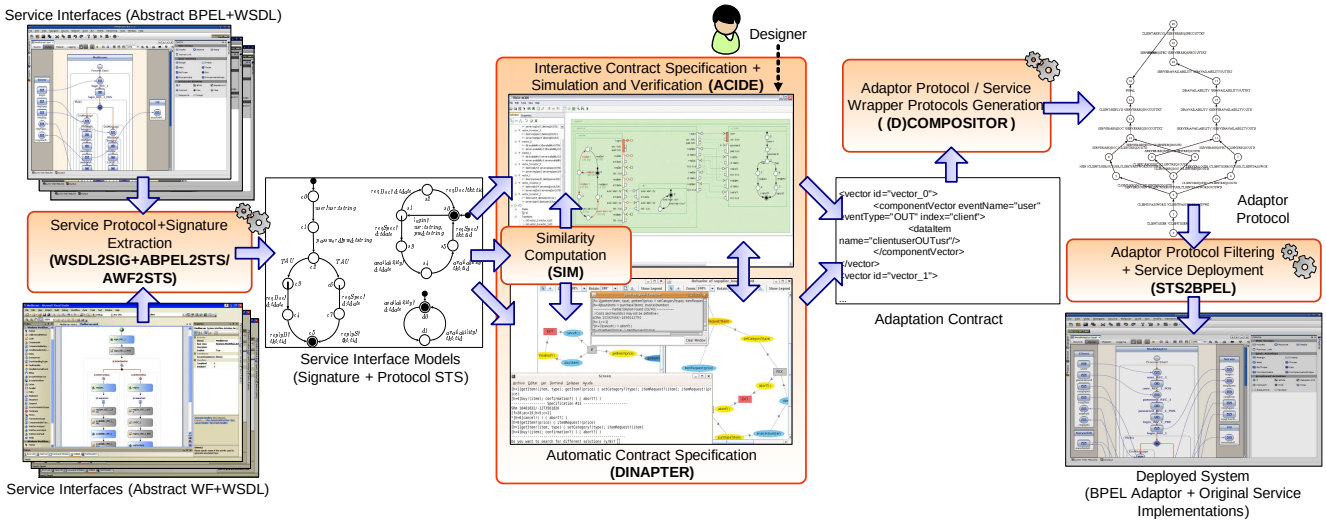


Figure 1. Adaptation process overview in ITACA

5. Concluding Remarks

Software adaptation is a satisfactory solution to build new systems involving reusable software services that present some mismatch cases in their interfaces. However, this is an error-prone task and therefore must be automated as much as possible. In this work, we have presented our approach to software adaptation and we focused on the adaptation contract specification, the only step of our proposal which requires human intervention. To help the designer in this task, we have proposed two alternative solutions to the manual design of contracts, which rely on graphical notation, interactive environment, and automatic generation techniques. In this work, we have also introduced what is, to the best of our knowledge, the first toolbox (ITACA) that fully supports a generative adaptation approach from beginning to end. ITACA supports the specification and verification of adaptation contracts, automates the generation of adaptor protocols, and relates our abstract models with implementation languages.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), and project P06-TIC-02250 funded by the *Junta de Andalucía*.

References

[1] L. de Alfaro, T. Henzinger, Interface Automata, in: Proc. of ESEC/FSE'01, ACM Press, 2001, pp. 109–120.

[2] S. Uchitel, M. Chechik, Mergin Partial Behavioural Models, in: Proc. of FSE'04, ACM Press, 2004, pp. 43–52.

[3] A. Basu, M. Bozga, J. Sifakis, Modeling Heterogeneous Real-time Components in BIP, in: Proc. of SEFM'06, IEEE Computer Society, 2006, pp. 3–12.

[4] M. Autili, P. Inverardi, A. Navarra, M. Tivoli, SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems, in: Proc. of ICSE'07, IEEE Computer Society, 2007, pp. 784–787.

[5] A. Brogi, R. Popescu, Automated Generation of BPEL Adapters, in: Proc. of ICSOC'06, Vol. 4294 of LNCS, Springer, 2006, pp. 27–39.

[6] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, F. Casati, Semi-Automated Adaptation of Service Interactions, in: Proc. of WWW'07, ACM Press, 2007, pp. 993–1002.

[7] A. Bracciali, A. Brogi, C. Canal, A Formal Approach to Component Adaptation, Journal of Systems and Software 74 (1) (2005) 45–54.

[8] C. Canal, P. Poizat, G. Salaün, Model-Based Adaptation of Behavioural Mismatching Components, IEEE Transactions on Software Engineering 34 (4) (2008) 546–563.

[9] M. Dumas, M. Spork, K. Wang, Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation, in: In Proc. of BPM'06, Vol. 4102 of LNCS, Springer, 2006, pp. 65–80.

[10] R. Mateescu, P. Poizat, G. Salaün, Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques, in: Proc. of ICSOC'08, LNCS, Springer, 2008, pp. 84–99.

[11] H. Foster, S. Uchitel, J. Kramer, LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography, in: Proc. of ICSE'06, ACM Press, 2006, pp. 771–774.

[12] X. Fu, T. Bultan, J. Su, Analysis of Interacting BPEL Web Services, in: Proc. of WWW'04, ACM Press, 2004, pp. 621–630.

[13] G. Salaün, L. Bordeaux, M. Schaerf, Describing and Reasoning on Web Services using Process Algebra, IJBPIM 1 (2) (2006) 116–128.

[14] J. Cubo, G. Salaün, C. Canal, E. Pimentel, P. Poizat, A Model-Based Approach to the Verification and Adaptation of WF/.NET Components, in: Proc. of FACS'07, Vol. 215 of ENTCS, Elsevier, 2007, pp. 39–55.

[15] M. Ouederni, Measuring Similarity of Service Protocols, Master Thesis, University of Málaga. Available on Meriem Ouederni's Webpage (Sep. 2008).

[16] A. Arnold, Finite Transition Systems, International Series in Computer Science, Prentice-Hall, 1994.

[17] J. A. Martín, E. Pimentel, Automatic Generation of Adaptation Contracts, in: Proc. of FOCLASA'08, ENTCS, 2008, to appear.

[18] J. Cámara, G. Salaün, C. Canal, M. Ouederni, Interactive Specification and Verification of Behavioural Adaptation Contracts, in: 9th International Conference on Quality Software (QSIC'09), IEEE, 2009, to appear.

[19] ISO/IEC, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, International Standard 8807, ISO (1989).

[20] R. Mateescu, M. Sighireanu, Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus, Science of Computer Programming 46 (3) (2003) 255–281.

[21] G. Salaün, Generation of Service Wrapper Protocols from Choreography Specifications, in: Proc. of SEFM'08, IEEE Computer Society, 2008, pp. 313–322.

[22] ITACA's Webpage, accesible from Javier Cámara's Webpage.

Runtime monitoring of contract regulated web services

Alessio Lomuscio[1], Wojciech Penczek[2,3], Monika Solanki[1] and Maciej Szreter[2]

[1] Department of Computing
Imperial College London, UK
[2] Institute of Computer Science
PAS, Poland
[3] University of Podlasie, Poland

Abstract. We investigate the problem of locally monitoring contract regulated behaviours in web services. We encode contract clauses in service specifications by using extended timed automata. We propose a *non intrusive* local monitoring framework along with an API to monitor the fulfilment (or violation) of contractual obligations. We illustrate our methodology by monitoring a service composition scenario from the vehicle repair domain, and report on the experimental results.

1 Introduction

Web services (WS) are now considered one of the key technologies for building new generations of digital business systems. Service level agreements (SLAs) provide a useful mechanism to establish agreed levels of service provision when interactions are invoked within certain parameters. Although SLAs are useful, they can represent only basic agreements of service provision. Applications running complex, human-like activities require more general and sophisticated declarative specifications certifying legal-like agreements among the parties.

A useful concept from the legal domain in this sense is the one of *contract* as found in human societies. Should a contract be broken by one of the parties, additional rights and/or obligations (e.g., penalties to be paid) may be applicable to some party. In this paper, we study the problem of monitoring runtime behaviours of *contract regulated web services*. While contracts are usually negotiated offline, it is of interest to monitor at runtime whether interactions between WS are complying to the contracts stipulated between the parties. We put forward a "symbolic" solution to the problem above. We represent both all possible behaviours and the contractually-correct ones as an appropriate timed automata [1] at local web-service level. Specifically we present a local contract runtime monitor (CRM) based on the symbolic toolkit Verics [5], a symbolic model checker for timed-automata. CRM checks the input at runtime against the symbolic representations provided, and reports to the service (or directly to the engineer) any mismatch, or *violation*, between the contract-compliant behaviours originally prescribed and the ones actually received in the input stream.

The significant advantage of the approach is that we do not need to keep the whole state space of the possible and the contract-compliant behaviours in memory but we can simply call the timed-automata engine at runtime to match moves against the stream of events coming from the input. The paper is structured as follows: In Section 2 we briefly

introduce the formalism of timed automata as used here. Section 3 presents our monitoring framework. We analyse a motivating case study in 4 and discuss the monitoring results. Section 5 presents related work and conclusions.

2 Monitoring via Timed Automata

Let $\mathbb{N}$ denote the set of naturals (including 0), $\mathbb{Z}$ - the set of integers, $\mathbb{Q}$ - the set of rational numbers, $\mathbb{R}$ ($\mathbb{R}_+$) - the set of (non-negative) reals, and V be a finite set of integer variables. By a *variable valuation* we mean any total mapping $\mathbf{v} : V \longrightarrow \mathbb{N}$. We extend the mapping $\mathbf{v}$ to expressions of $Ex(V)$ in the usual way. The satisfaction relation ($\models$) for the boolean expressions is also standard.

Given a variable valuation $\mathbf{v}$ and an instruction $\alpha \in Ins^L(V)$, we denote by $\mathbf{v}(\alpha)$ the valuation $\mathbf{v}'$, obtained after executing α at $\mathbf{v}$, which is formally defined as follows:

- if $\alpha = \epsilon$ then $\mathbf{v}' = \mathbf{v}$,
- if $\alpha = (v := ex)$, then $\mathbf{v}'(v) = \mathbf{v}(ex)$ and $\mathbf{v}'(v') = \mathbf{v}(v')$ for all $v' \in V \setminus \{v\}$,
- if $\alpha = \alpha_1\alpha_2$, then $\mathbf{v}' = (\mathbf{v}(\alpha_1))(\alpha_2)$.

Let $\mathcal{X} = \{x_1, \ldots, x_{n_\mathcal{X}}\}$ be a finite set of real-valued variables, called *clocks*. The set of *clock constraints* over $\mathcal{X}$ and V, denoted $\mathcal{C}(\mathcal{X}, V)$, is defined by the grammar: $\mathfrak{cc} ::= true \mid x_i \sim c \mid x_i \otimes x_j \sim c \mid x_i \otimes x_j \sim v \mid x_i \otimes v \sim c \mid v \otimes w \sim x_i \mid \mathfrak{cc} \wedge \mathfrak{cc}$, where $x_i, x_j \in \mathcal{X}$, $v, w \in V$, $c \in \mathbb{N}$, $\otimes \in \{+, -\}$, and $\sim \in \{\leq, <, =, >, \geq\}$. Let $\mathcal{X}^+$ denote the set $\mathcal{X} \cup \{x_0\}$, where $x_0 \notin \mathcal{X}$ is a fictitious clock representing the constant 0. A *clock-to-clock assignment* A over $\mathcal{X}$ is a function $A : \mathcal{X} \longrightarrow \mathcal{X}^+$. $Asg(\mathcal{X})$ denotes the set of all the assignments over $\mathcal{X}$. By a *clock valuation* we mean a mapping $\mathbf{c} : \mathcal{X} \longrightarrow \mathbb{R}_+$. The satisfaction relation ($\models$) for a clock constraint $\mathfrak{cc} \in \mathcal{C}(\mathcal{X}, V)$ under a clock valuation $\mathbf{c}$ and a variable valuation $\mathbf{v}$ is defined as:

- $(\mathbf{c}, \mathbf{v}) \models (x_i \otimes v \sim c)$ iff $\mathbf{c}(x_i) \otimes \mathbf{v}(v) \sim c$,
- the other cases are defined similarly.

In what follows, the set of all the pairs $(\mathbf{c}, \mathbf{v})$, composed of a clock and a variable valuation, satisfying a clock constraint $\mathfrak{cc}$ is denoted by $[\![\mathfrak{cc}]\!]$. Given a clock valuation $\mathbf{c}$ and $\delta \in \mathbb{R}_+$, by $\mathbf{c} + \delta$ we denote the clock valuation $\mathbf{c}'$ such that $\mathbf{c}'(x) = \mathbf{c}(x) + \delta$ for all $x \in \mathcal{X}$. Moreover, for a clock valuation $\mathbf{c}$ and an assignment $A \in Asg(\mathcal{X})$, by $\mathbf{c}(A)$ we denote the clock valuation $\mathbf{c}'$ such that for all $x \in \mathcal{X}$ we have $\mathbf{c}'(x) = \mathbf{c}(A(x))$ if $A(x) \in \mathcal{X}$, and $\mathbf{c}'(x) = 0$ if $A(x) = x_0$. Finally, by $\mathbf{c}^0$ we denote the *initial* clock valuation, i.e., the valuation such that $\mathbf{c}^0(x) = 0$ for all $x \in \mathcal{X}$. In this paper we assume a slightly modified definition of timed automata with discrete data [17], which extend the standard timed automata of Alur and Dill in the following way:

Definition 1. *A* timed automaton with discrete data *(TADD) is a tuple* $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$, *where*

- Σ *is a finite set of* labels (actions),
- L *is a finite set of* locations,
- $l^0 \in L$ *is the* initial location,
- V *is the finite set of integer variables,*
- $\mathcal{X}$ *is the finite set of clocks,*

– $\mathcal{E} \subseteq L \times \Sigma \times Bool(V) \times \mathcal{C}(\mathcal{X}, V) \times Ins^L(V) \times Asg(\mathcal{X}) \times L$ *is a* transition relation*, and*
– $\mathcal{I} : L \longrightarrow \mathcal{C}(\mathcal{X}, \emptyset)$ *is an* invariant function*.*

The invariant function assigns to each location a clock constraint (without integer variables[4]) expressing the condition under which $\mathcal{A}$ can stay in this location.

The semantics of a TADD $\mathcal{A}$ is given below.

Definition 2. *The* semantics *of* $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$ *for an initial variable valuation* $\mathbf{v}^0 : V \longrightarrow \mathbb{Z}$ *is a labelled transition system* $\mathcal{S}(\mathcal{A}) = (Q, q^0, \Sigma_{\mathcal{S}}, \longrightarrow)$*, where:*

– $Q = \{(l, \mathbf{v}, \mathbf{c}) \mid l \in L \wedge \mathbf{v} \in \mathbb{Z}^{|V|} \wedge \mathbf{c} \in \mathbb{R}_+^{|\mathcal{X}|} \wedge \mathbf{c} \models \mathcal{I}(l)\}$ *is the set of states,*
– $q^0 = (l^0, \mathbf{v}^0, \mathbf{c}^0)$ *is the initial state,*
– $\Sigma_{\mathcal{S}} = \Sigma \cup \mathbb{R}_+$ *is the set of labels,*
– $\longrightarrow \subseteq Q \times \Sigma_{\mathcal{S}} \times Q$ *is the smallest transition relation:*
  - *for* $a \in \Sigma$*,* $(l, \mathbf{v}, \mathbf{c}) \xrightarrow{a} (l', \mathbf{v}', \mathbf{c}')$ *iff there exists a transition* $t = (l, a, \beta, \mathfrak{cc}, \alpha, A, l') \in \mathcal{E}$ *such that* $\mathbf{v} \models \beta$*,* $(\mathbf{c}, \mathbf{v}) \models \mathfrak{cc}$*,* $\mathbf{v}' = \mathbf{v}(\alpha)$*,* $\mathbf{c} \models \mathcal{I}(l)$*, and* $\mathbf{c}' = \mathbf{c}(A) \models \mathcal{I}(l')$ (action transition)*,*
  - *for* $\delta \in \mathbb{R}_+$*,* $(l, \mathbf{v}, \mathbf{c}) \xrightarrow{\delta} (l, \mathbf{v}, \mathbf{c} + \delta)$ *iff* $\mathbf{c} \models \mathcal{I}(l)$ *and* $\mathbf{c} + \delta \models \mathcal{I}(l)$ (time transition)*.*

Intuitively, in the initial state all the variables are set to their initial values, and all the clocks are set to zero. Then, at a state $q = (l, \mathbf{v}, \mathbf{c})$ the system can either execute an action or time transition.

2.1 TADD Semantics for RMCS

Inspired by related work in the formal representation of states of compliance and violation [10], we partition the set of global states Q of $\mathcal{S}(\mathcal{A})$ for $\mathcal{A} = (\Sigma, L, l^0, V, \mathcal{X}, \mathcal{E}, \mathcal{I})$ into two subsets G and R such that $G \cap R = \emptyset$[5]. The set G represents *green* (or *ideal*) states, whereas R represents the *red* (or *non-ideal*) ones. Intuitively, G contains the states of compliance and R contains the states of violation with respect to the contract, i.e., the whole set of clauses being included. Figure 1 illustrates the intuition behind the semantics.

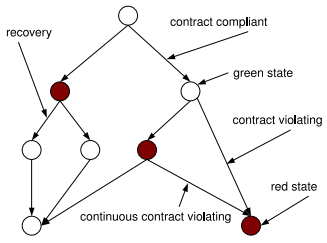


Fig. 1. Partitioning of states and transitions in a TADD

Based on the above partitioning each action transition (q, a, q') of $\mathcal{S}(\mathcal{A})$ can be one of the following four types of transitions:

[4] To ensure the monotonicity of the timed successor relation.
[5] This partition is obtained "location-wise" from a partition of the set of locations L of $\mathcal{A}$.

– **Contract compliant**: between green and green states, i.e., $q, q' \in G$. These transitions occur when the observed behaviour is in compliance with the prescribed behaviour of the contract.
– **Contract violating**: between green and red states, i.e., $q \in G$ and $q' \in R$. These transitions occur when the observed behaviour violates the prescribed behaviour of the contract.
– **Recovery**: between red and green states, i.e., $q \in R$ and $q' \in G$. These transitions occur when a recovery action is taken by the service after a violation of the prescribed behaviour is recorded.
– **Continuous contract violating**: between red and red states, i.e., $q, q' \in R$. The transitions occur when no recovery results from a previous violation.

We say that there is a *step* from state q_1 to q_2 in $\mathcal{A}$ if $q_1 \xrightarrow{\delta_1} q_1' \xrightarrow{a} q_2' \xrightarrow{\delta_2} q_2$, for some states $q_1', q_2' \in Q$, $\delta_1, \delta_2 \in \mathbb{R}_+$, and $a \in \Sigma$.

3 Runtime monitoring framework

Our architecture for local monitoring, RMCS, is illustrated in Figure 2.

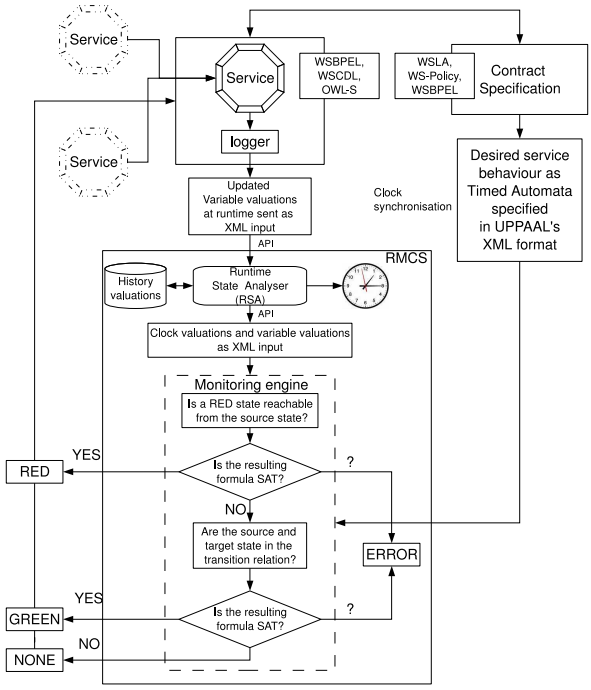


Fig. 2. The general architecture and methodology

Agents implementing WS are the primary entities within our framework. Service behaviour and contracts associated with them may be specified at a high level using WS

standards, e.g., WSBPEL [13] and contracts, e.g., WSLA [7]. The TADD specification for the service is engineered from these interface representations. The specification of service behaviour used by RMCS is the TADD representation described in Section 2. We use the XML format generated by the model checker UPPAAL for representing the TADD. The runtime state analyser interfaces with the logger for receiving snapshots of the latest variable valuations generated by the service. Snapshots are passed to the RSA via the logging framework. RSA is also responsible for updating clocks by querying the system hardware, in accordance with the granularity of a *tick* chosen by the service. The monitoring engine is the core component responsible for testing the conformance of runtime service behaviour presented as an input from the RSA, against the prescribed TADD specification of the service.

Each execution step passed to the engine is encoded and its conformance to the TADD specification is tested. Our SAT-based verification method does not need to construct the complete model for $\mathcal{A}$, which could be unfeasible for both the explicit-state as well as BDD-based methods. The engine checks at runtime whether the stream of execution steps received as inputs from the RSA, conforms with its symbolic representation of all possible behaviours. For each execution step, the answer returned by the monitoring engine is one of the following facts:

- **GREEN** - the step is conforming with the specification, i.e., there is a contract compliant transition between the source and target states.
- **RED** - a red state is reached as a target of the transition given, i.e., a contract has been violated as a result of the transition. This also signifies the fact that the inputs do not comply with the extended format of the TADD for the service.
- **NONE** - the step is not conforming with the specification, i.e., there is no such transition, neither contract compliant or otherwise.
- **ERROR** - the specification given does not mirror the observed transition so it amounts to an error.

Results reported at runtime may be analysed in several ways.

4 A vehicle repair contract: case study

We consider a service composition scenario that defines a repair contract between a client (C) and a vehicle repair company (RC). A repair contract specifies details concerning a particular repair, i.e., the type of repair to be performed, price, dates, pickup and delivery locations etc. For simplicity we only model the behaviour of RC. Table 1 identifies some of the contract clauses governing the actions taken by RC, the deadlines against which the contracts are monitored, if the clause can be violated, and, if a violation is recorded, whether any recovery is possible. Note that in some cases RC may take an "offline" action, in response to a violation from which no recovery may be possible. For example consider clause 6: "For any violation take recovery action within 3 days". If the recovery action is not taken, C may take an offline legal action against RC.

The informal behaviour of RC is described as follows. When RC receives a request from C to undertake a repair job, it sends a repair proposal. In response, C sends an acceptance or rejection message. If accepted, RC sends a contract initiation message to C. RC then waits for the vehicle to arrive, failing which it sends two reminders to C. If the vehicle fails to arrive, it takes an offline action. As per the contract, RC is *obliged* to

clause	Contract regulated actions	Deadline	Violation	Recovery
1	Receives a repair request by C	5 days	-	-
2	Sends a repair proposal to C	7 days	-	-
3	Assess damage to the vehicle	3 days	yes	yes
4	Execute repair	30 days	yes	yes
5	Send repair report to C	5 days	yes	yes
6	For any violation take recovery action	3 days	yes	no (take offline action)

Table 1. Some contract regulated actions for RC

assess the damage, repair the vehicle and send a report to C. On receiving the report, C is *obliged* to send payment to RC. If the payment is not sent, RC sends two reminders to C and then takes an offline action.

The actions taken by RC in response to messages sent by C are monitored to meet the deadlines set for various activities as per the contract. Failure to meet deadlines is considered a violation of the contractual obligations. In some cases a recovery from the violation may be possible.

4.1 Monitoring the runtime behaviour of the Repair Company

The full set of behaviours of the repair company is represented by a TADD[6]. As described in Section 4, deadlines for various activities are decided during contract negotiation between the parties. Deadlines are defined in terms of number of days. For example consider a contract clause to be monitored: *If C sends a damaged vehicle to RC, RC assesses the damage to the vehicle within 3 days -clause (3) in table 1. A snippet of the TADD for the clause is shown in the Figure 3.* Figure 3 describes the timeline in number

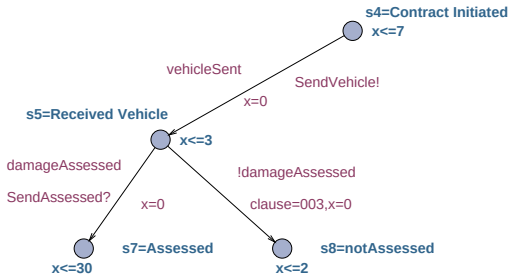


Fig. 3. TA specification of clause (3)

of days for clause (3), a snapshot passed to RSA at $x = 0$ from the logger when a vehicle for repair arrives, snapshots sent to the monitoring engine by the RSA and the results from monitoring. As per the contract, once a damaged vehicle has arrived the damage has to be assessed within 3 days. A snapshot is again sent by the logger to the RSA at $x = 5$. The snapshot taken at $x = 0$ and at $x = 5$ are sent by the RSA as a pair - or as a

[6] The complete TADD for the example is too large to be shown here.

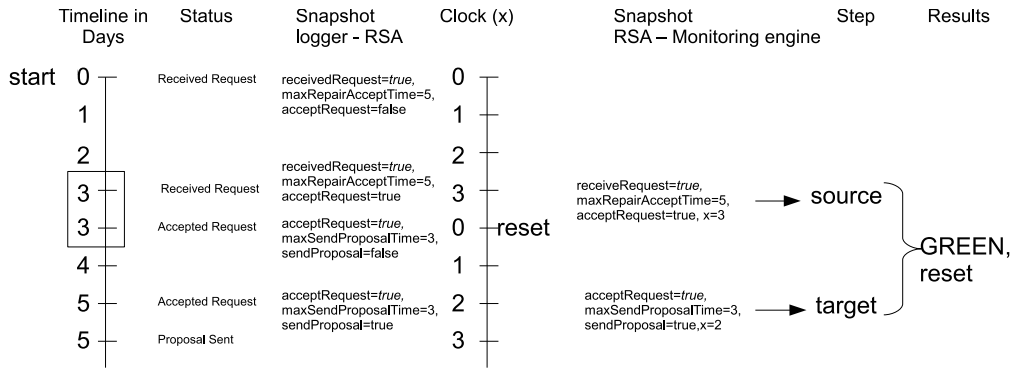


Fig. 4. Runtime valuations for clause (2)

"step" to RMCS. The results returned by the monitoring engine are $\{RED, reset, 003\}$. RED signifies that a violation has occurred, i.e., the damage was not assessed within the deadline, $reset$ indicates that the clock has been reset and 003 indicates the clause index that has been violated.

4.2 Experimental results and Discussion

In order to validate our methodology, we implemented the above case study and monitored several runtime execution steps for the service. RMCS successfully monitored 8 execution steps per second depending on the number of variables defined for the steps. Violated contracts and clock resets were reported by RMCS. Note that this could be useful in the context of monitoring SLAs, where typically large number of execution steps need to be monitored to ensure a reliable Quality-of-Service.

5 Related work and conclusions

In this paper we presented a symbolic approach based on timed automata for the runtime monitoring of contract regulated agent based WS. Monitoring service behaviour has been an active area of research. Several efforts have investigated various formalisms and frameworks for the monitoring of functional and non-functional properties of services. The monitoring problem has been considered for several formalisms in papers [16, 2, 4, 14, 12, 11, 9, 3]. Timed automata have been used in earlier work such as [8] on monitoring and fault diagnosis of systems, while [15] presents an approach which also uses timed automata for monitoring SLAs. The aims of the above approaches are however quite different from our objectives in this paper. However [8, 15] are not concerned with local monitoring of contract-based executions.

An attractive feature of our approach over those mentioned above is that histories and pending contracts are not stored in memory during the monitoring. This positively impacts the scalability of the approach and is particularly useful when monitoring multiple and long running contracts between several services. As a case study we presented the monitoring of contracts for a repair company. Although the TADD for the service

is not large enough to exploit the full capabilities of RMCS, we believe it is still sufficiently significant to demonstrate the methodology and scope of the proposed approach. Experiments demonstrate larger scenarios would be handled just as well by the technique.

Much work remains to be done. An important part of our future work is the translation to TADDs from high level specification standards such as WSBPEL.

References

1. R. Alur. Timed Automata. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.
2. Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*.
3. Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing*. ACM.
4. Domenico Bianculli and Carlo Ghezzi. Monitoring conversational web services. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*. ACM.
5. P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, LNCS. Springer-Verlag.
6. N. Eén and N. Sörensson. MiniSat. http://minisat.se/MiniSat.html.
7. Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *J. Netw. Syst. Manage.*, 2003.
8. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop on Model Checking of Software (SPIN'04), Barcelona, Spain*, LNCS.
9. Zheng Li, Yan Jin, and Jun Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC '06: Proceedings of the Australian Software Engineering Conference (ASWEC'06)*. IEEE Computer Society.
10. A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.
11. G. Mahbub, K.; Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: initial implementation and evaluation experience. In *ICWS'05, IEEE International Conference on Web Services*.
12. Carlos Molina-Jimenez, Santosh Shrivastava, Ellis Solaiman, and John Warne. Contract representation for run-time monitoring and enforcement. *cec*, 2003.
13. OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0, 2007.
14. Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.
15. F. Raimondi, J. Skene, L. Chen, and W. Emmerich. Efficient monitoring of web service slas. Technical report, UCL, London, 2007.
16. Monika Solanki. *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Service*. PhD thesis, De Montfort University, Leicester, UK, October 2005.
17. A. Zbrzezny and A. Półrola. SAT-based reachability checking for timed automata with discrete data. *Fundamenta Informaticae*, 79(3-4):579–593, 2007.

Towards a Rigorous IT Security Engineering Discipline

Antonio Maña.
Computer Science Department.
University of Málaga. Spain.
amg@lcc.uma.es

Abstract. Current practices for developing secure systems are still closer to art than to an engineering discipline. Security is still treated too frequently as an add-on and is therefore not integrated into IT systems development practices and tools. Experienced security artisans continue to be the key for achieving acceptable levels of security in IT systems. In this situation we could safely state that security engineering is a hype. In fact, the term security engineering has been used to denote partial approaches that cover only small parts of the processes that are required in order to create a secure system, like modelling, verification, programming, etc. Moreover, one finds in the literature that the main books about security engineering describe threat-based engineering approaches. The main drawbacks of these approaches is that they fail to provide a reasonable level of support for systematic engineering since the identification, characterization and specification of the requirements (frequently based on avoiding threats) as well as the selection of appropriate mechanisms and countermeasures depends on the experience of the engineers. Consequently, these approaches are inherently opposed to what an engineering discipline should be and represent only minor improvements over the security art. This paper discusses this vision and advocates a change of paradigm based on rigour and precision. In particular, the paper presents an approach based on the precise and formal specification of both security requirements and security solutions as the basis for the engineering of secure systems and discusses the role of contracts in security engineering.

Introduction

Current practices for developing secure systems are still closer to art than to an engineering discipline. IT Security is treated as an add-on and is therefore not integrated into software development practices and tools. With more and more aspects of our lives being affected by computing systems and with the inevitable trend towards IT systems in which humans are immersed, the aspects of assurance and certification become crucial. However, we still depend on the knowledge of experienced security artisans in order to predict the threats that the system will have to withstand and to prevent them, achieving acceptable levels of security, and this art-like approach does not allow us to have guarantees and proofs of the security of those systems. Moreover, the growing complexity of these systems is becoming larger than the capacity of humans to understand and secure an IT system.

The existence of an IT security engineering discipline could not only change that situation of lack of guarantees but also improve the security of IT systems by allowing them to be prepared to operate in unforeseen contexts and to be able to provide security to the extremely complex and dynamic IT systems that are coming in the near future.

Traditionally, the term security engineering has been used to denote partial approaches that cover only small parts of the processes that are required in order to create a secure system, like modelling, verification, programming, etc. Several approaches and research strands have tried to address this situation in order to introduce rigour and engineering approaches in the treatment of security aspects in information systems, mainly focusing on the development phases. Paradoxically, many of the best known initiatives for enhancing the security of computer systems have been based on guidelines, recommendations, best practices, certification and similar approaches lacking the necessary rigour and precision that one would expect when dealing with "engineering" and even more with "security engineering". Some examples are: Common criteria [1], traditional security patterns [2], Sarbanes-Oxley [3] and HIPA [4] Acts, etc.

Even in the cases of approaches that are closer to a methodology and have achieved a certain level of maturity, the key concepts and workflows are highly influenced by the way security has been treated by the security artisans. Therefore, one finds in the literature that the main books about security engineering describe threat-based engineering approaches. The main drawbacks of these approaches is that they fail to provide a reasonable support for systematic engineering since the identification, characterization and specification of the threats as well as the selection of appropriate mechanisms and countermeasures depends on the experience of the engineers. Consequently, these approaches represent only minor improvements over the security art. Yet, they have been used for some time with uneven popularity and results.

In this paper we discuss this aspect and advocate a change of paradigm based on the precise and formal specification of security solutions and security properties and the use of these formally verified properties as the basis for the expression of requirements and the engineering of secure systems. The paper shows that threat-based security engineering (i) is the origin of penetrate-and-patch situation; (ii) does not result in specifications that can survive evolution of systems and context; and (iii) does not capture the requirements, but the mechanisms selected to achieve them, which results in poor maintainability of the systems produced. The paper also introduces two pillars to solve the situation. On the one hand, we present the SERENITY model of secure and dependable systems and show how it supports the creation of secure and dependable systems for these new computing paradigms. On the other hand we discuss the concept of contract and the role it plays in ensuring a rigorous treatment of security. Finally some conclusions will be drawn and a proposal for establishing a renewed security engineering discipline will be advocated.

Threat-based security engineering considered harmful

Today, the current trend towards distributed and open systems, has revealed the important limitations of threat-based security engineering. The main problems that the new computing paradigms introduce are the high levels of heterogeneity, dynamism and autonomy, as well as the large scale and high complexity. The result is that engineers have to deal with runtime situations that are unpredictable at design time, which are very difficult if not impossible to secure using threat-based or attack-based approaches. In particular, threat-based security engineering creates systems that are very context-dependent, and therefore, fail to address the needs of the future open and distributed systems paradigms. This approach was acceptable and even successful in the times of closed and static systems, running on homogeneous and well-known platforms, because the runtime context was more or less predictable and therefore a security expert could possibly identify the threats that the IT system under development had to withstand and could find appropriate solutions to avoid them. However, once we face the challenges introduced by open systems, which will be built

from components at runtime; dynamic evolution, which increases unpredictability and introduces the need for security solutions to adapt to the system evolution; and extremely heterogeneous platforms, which essentially makes impossible to rely on infrastructure security, we are obliged to admit that we have reached the end of the threat-based approaches and we need to use a new approach that can deal with the aforementioned challenges.

Some of the main reasons why we consider that threat-based security engineering is not a valid approach to provide acceptable levels of security for the future IT systems are:

- **Lack of dynamism and support for evolution.** Threats change when systems change even if the protection goals remain the same. Therefore, threat-based systems require full reengineering to cope with any context change. Moreover, it becomes very difficult to identify the changes required in the system, as a result of a change in the context.
- **Poor traceability.** Threats are difficult to trace to protection goals or security requirements. This is precisely the reason why systems engineered following a threat-based approach tend to be extremely difficult to maintain and to adapt to new context conditions.
- **Expression of user requirements is lacking, not precise or context-dependent.** This is the one of the main weakness and is related to some of the others, and in particular to the previous two. When the input to design process is expressed in terms of threats and attacks, the user requirements become lost (or at least hidden) and consequently the system under development becomes weaker in terms of maintainability, traceability and resilience to evolution. Moreover, the longevity and stability of the specification of the system are reduced.
- **The complete set of threats is impossible to identify.** The impossibility to guarantee the completeness of the set of identified threats, has been traditionally a major weakness of the threat-based engineering approaches. Given the unpredictability that characterizes the future computing paradigms, it will become impossible to identify not even a small fraction of the potential threats, even for the most experienced and visionary security experts.
- **The threat-based approach is closely related to the penetrate-and-patch vicious cycle.** As it is impossible to predict at development time all attacks and threats that the system will face while in operation, the new threats and vulnerabilities that are discovered during the system operation require the system to be patched, which inevitably leads to a degradation of the quality and the introduction of new vulnerabilities.
- **Assurance and certification can not be based on threats.** This aspect is specially important in security critical systems, but also in all socio-technical systems. Stating (or even proving) that a system can withstand a threat does not say much about what can be guaranteed about the system.
- **Poor user communication.** Tell your customers that their system will be secure because you will implement state-of-the-art protection against cross-site scripting and avoid eavesdropping by using authenticated TLS connections. They may be happy to hear some fashionable buzzwords, but it is most likely that they do not understand if what you are providing solves their problem. Some advances based on the specification of abuse and misuse cases can help, but still they lack precision and do not provide the guarantees that your customer would need.

In consequence, we believe that the threat-based security engineering era has come to an end and it is time to find an appropriate replacement. The next two sections will present two

pillars of this replacement.

The SERENITY model for IT security engineering

The goal of this section is to facilitate the understanding of the SERENITY approach to secure IT systems engineering. However, due to its size and complexity, it is out of the scope of this paper to provide a complete view of SERENITY. The interested reader is referred to reference [5] for a comprehensive description of the whole system. The SERENITY project has produced two main results related to our discussion:

A model of secure system engineering that has two main features:

- *Separation between security solution development and application development.* In this way security experts develop, analyse and characterize security solutions and components while application developers and programmers concentrate on developing applications without the need to implement security solutions (which is normally a recipe for disaster given the lack of security expertise of average application developers). All application developers have to know about security is to express their security requirements and to include references to the security solutions (which are precisely and semantically described using a series of modelling artefacts called S&D Artefacts, as we will show below) that they need, in their application code. The more abstract the level of the artefact selected at development-time is, the more the flexibility for the selection of the specific implementation of the solution to use at runtime.
- *Run-time support* enabling the dynamic selection, adaptation and substitution of security solutions at run-time. Additionally, at run-time the selected solutions are monitored to ensure their correct operation. To achieve this, it is necessary to have a framework supporting the management of a library of available security solutions and the constant evolution of applications based on such solutions, taking into account the context in which they are applied. The SERENITY Runtime Framework (SRF) is the tool that provides this support. It is able to select the most appropriate S&D solution among the ones available, based on the end-user requirements and the actual runtime context.

A series of modelling artefacts used to capture knowledge about different security aspects, such as properties, services, and solutions.

In SERENITY, the main pillar of building secure and dependable solutions are the enhanced concept of S&D Properties and S&D Pattern. An **S&D Property** is a precise specification of one or more security goals that can be applied to any computational object. It is important to highlight that, opposed to the common belief, the number of security properties (i.e. confidentiality, integrity, authenticity, non-repudiation...) is not limited. We could express this more precisely saying that, even if we accept that the number of "abstract properties" could be considered limited, the number of interpretations (semantics) of these abstract properties is unbounded. The SERENITY model is based on the assumption that there will be different definitions (corresponding to interpretations) for the security properties. There are many reasons supporting this assumption: for instance, legislation, cultural etc. Under this assumption, and in order to guarantee the interoperability of systems based on those properties and of solutions fulfilling the properties, we need to be able to precisely express the interpretation (semantics) of each property and to exploit the relations between different properties. For a more complete description of the concept of S&D Property, the reader is referred to references [5] and [6].

An S&D Pattern captures security expertise in a way that, in our view, is more appropriate than other related concepts because it focuses on precision and well-defined semantics. Furthermore, because secure interoperability is an essential requisite for the widespread adoption of the SERENITY model, trust mechanisms are a central aspect of S&D Patterns. SERENITY S&D Patterns include a precise functional description of the mechanisms they represent, references to the S&D properties provided, constraints about the context that is required for deployment, and specifications of how to adapt and monitor the mechanisms, as well as trust mechanisms designed to (i) guarantee the origin and integrity of the descriptions contained in the different SERENITY artefacts; and (ii) support the evolution of artefacts and mechanisms and the maintenance of SERENITY systems.

The concept of S&D Pattern has been materialized in a series of modelling elements that we call S&D Artefacts. Abstract S&D solutions are represented by three main artefacts: S&D Classes, S&D Patterns and S&D Implementations. A fourth element, called Executable Component is part of the SERENITY artefacts, but in this case it is not a modelling artefact but an operational one. The above artefacts are briefly described as follows:

- **S&D Patterns** represent abstract S&D solutions. These solutions are well-defined mechanisms that provide one or more S&D Properties. The popular SSL protocol is an example of an S&D solution that can be represented as an S&D Pattern. One important aspect of the solutions represented as S&D Patterns is that they can be statically analysed (e.g. using verification tools based on formal methods). However, the limitations of the static analysis and the assumption that perfect security is not achievable, introduce the need to support the dynamic validation of the described solutions by means of monitoring mechanisms.
- **S&D Classes** represent S&D services (abstractions of a set of S&D Patterns characterized for providing the same S&D Properties and being compatible with a common interface). An example of an S&D Pattern Class could be a *Confidential Communication Class*, which could define an interface including for instance, an abstract method *SendConfidential(Data, Recipient)*. The purpose of introducing this artefact is to facilitate the dynamic substitution of the S&D solutions at runtime. This approach allows us to create an application bound to an S&D Class. Then, all S&D Patterns (and their respective S&D Implementations) belonging to an S&D Class will be selectable by the framework at runtime to serve that application.
- **S&D Implementations** represent operational S&D solutions, which are in turn called Executable Components. It is important to note that the expression "operational solutions" refers here to any final solution (e.g. component, web service, library, etc.) that has been implemented and tested. These solutions are made accessible to applications thanks to the SERENITY Runtime Framework (SRF). The description of either a specific dynamic library providing encryption services or a web service providing time-stamping services, are examples of S&D Implementations. S&D Implementations capture implementation-specific features, such as performance, platform, programming language or any other feature not fixed at pattern's level. The OpenSSL implementation of the SSL protocol ca be described using an S&D Implementation.

These new concepts can be useful in two different ways: at design/deployment time and at run time. In the first case, we must consider that today's large applications are built by integrating solutions from different sources and at different levels of abstraction. These applications face the existence of threats and errors that may require us to perform maintenance on them. Readers can find details on the implementation of SERENITY applications and S&D solutions in references [7] and [8]. By using the SERENITY approach

we allow that this maintenance is performed with a minimum effort, even automatically and seamlessly in some cases. In the second case (i.e., during runtime) S&D Patterns are used in order to support automated adaptation of the S&D solutions to the changing context conditions.

The role of contracts in a rigorous security engineering

Informally speaking we could define a contract as an agreement between two or more parties that establishes obligations for these parties and guarantees about those obligations. More precisely, the *BusinessDictionary.com* definition is a "Voluntary, deliberate, and legally enforceable (binding) agreement between two or more competent parties." The main difference between the two is that we do not necessarily assume that a contract has to have any legal meaning (although we do not exclude that possibility). Contracts can be used in IT security for different purposes. Among these, we highlight the following:

- **Contracts as means for agreeing on security aspects.** In this case the contract establishes the terms (e.g. mechanisms, guarantees, referees and trusted third parties, etc.) that will be used in an interaction between two or more parties. A well-known example of this is constituted by SLAs (service-level agreements) that establish aspects related to the QoS (Quality of Service) like bandwidth, uptime, throughput, etc. Also in this category we find expression of the "terms of use".
- **Contracts as specifications.** A contract can be used to specify aspects of the operation of an entity. For instance, it can be used to specify the means by which the entity ensures the confidentiality of the data processed in an application in cloud computing or service-oriented computing. Another interesting case in this category is that of software contracts, which have been used in component-based development and especially for COTS (components-off-the-shelf). The same concept has recently been applied to the field of secure coding. In fact, some mature development strategies like PCC (proof carrying code) are closely related to this. In PCC, executable code comes with proofs that demonstrate adherence to a contract. These proofs can be verified by the runtime environment prior to code execution.
- **Contracts as guarantees.** A contract can be used to state guarantees about the operation of an entity. For instance, it can be used to guarantee that an economic compensation will be available to the user should the confidentiality of the data processed in an application in cloud computing or service-oriented computing be broken.
- **Contracts as disclaimers.** The idea in this case is to make the user aware of the risks that the software introduce and to declare the limitations of the guarantees of a provider.

From the previous descriptions one can easily understand that the concept of contract constitutes a key element in providing precision, control, limitations and rigour into the security engineering discipline. Contracts reduce uncertainty and provide support for sound reasoning about dynamic, distributed and composed systems.

In the SERENITY model, contracts are mainly used to establish agreements between different SRFs, but the contents of the descriptions made using the S&D Artefacts can also be considered as contracts. In the first case, the SRF exposes a negotiation interface for external systems. The negotiation interface is used in order to establish the configuration of the interacting SRFs when two applications supported by different SRFs need to communicate using the same S&D Solution. In the second case, S&D Artefacts include means for expressing contractual facts (such as guarantees, SLAs, terms of use and disclaimers) as well as trust mechanisms designed to guarantee their origin and integrity by

means of digital signatures. These trust mechanisms can also be used to give validity to the contractual facts included in the artefact description.

Conclusions

In this paper we have discussed the drawbacks of threat-based security engineering and have advocated a change of paradigm based on the precise and formal specification of security solutions and security properties and the use of these formally verified properties as the basis for the expression of requirements and the engineering of secure systems. We have shown that threat-based security engineering (i) is the origin of penetrate-and-patch situation; (ii) does not result in specifications that can survive evolution of systems and context; and (iii) does not capture the requirements. We have introduced two pillars to solve the situation: the SERENITY model of secure and dependable systems and the concept of contract and the role it plays in ensuring a rigorous treatment of security.

We firmly believe that the trends that are driving the future computing scenarios are incompatible with threat-based security engineering, and therefore we propose to establish new models and principles that can result in a change of paradigm. The ultimate goal is to establish IT security as a fully fledged engineering discipline, based on the definition of integrated processes with well-defined goals and interfaces that combine the different techniques, methodologies and tools to support the engineering of future secure IT systems.

References

[1] Common Criteria Editorial Board. Common Criteria for Information Technology Security Evaluation. Version 3.1 Revision 1. September 2006. Available from http://www.commoncriteriaportal.org

[2] M. Schumacher, E. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad. Security Patterns - Integrating Security and Systems Engineering. John Wiley & Sons, 2005.

[3] Congress of the United states of America. Sarbanes-Oxley Act of 2002. Available from http://www.access.gpo.gov

[4] Congress of the United states of America. Health Insurance Portability and Accountability Act of 1996. Available from http://www.hhs.gov/ocr/hipaa

[5] G. Spanoudakis, A. Maña and S. Kokolakis (2009). "Security and Dependability for Ambient Intelligence". Advances in Information Security, ISBN 978-0-387-88775-3. Springer.

[6] A. Maña, G. Pujol (2008). Towards formal specication of abstract security properties. Third International Conference on Availability, Reliability and Security (ARES2008), Barcelona, March 2008. IEEE Computer Society Press.

[7] F. Sanchez-Cid, A. Maña (2008). SERENITY Pattern-based Software Development Life-Cycle. 2nd International Workshop on Secure systems methodologies using patterns (SPATTERN'08). IEEE Computer Society.

[8] D. Serrano, A. Maña, and A. D. Sotirious (2008). Towards precise and certified security patterns. 2nd International Workshop on Secure systems methodologies using patterns (SPATTERN'08). IEEE Computer Society.

Models for Open Transactions*

Roberto Bruni and Ugo Montanari

Computer Science Department, University of Pisa, Italy

Abstract. Loosely coupled interactions permeate modern distributed systems, where autonomous applications need to collaborate by dynamical assembly. We can single out three different phases occurring in every collaboration: 1) negotiation of some sort of contracts, mediating the needs of prospective participants; 2) acceptance or rejection of the contract; 3) contract-guarantee execution. The above scheme, called NCE for short (Negotiation, Commit, Execution), covers a wide range of situations, ranging from sessions and transactions to proof-carrying code. In the paper we concentrate on the notion of open transaction and on Zero-Safe Nets, a model developed by the authors for modelling long transactions. We extend the latter to cover the three-phase process above.

1 Introduction

Recent years have witnessed a progressive shifting of importance from the traditional concept of safe, overly controlled computation to open-ended, loosely coupled interactions, which permeate modern distributed systems. In particular, features such as concurrency, dynamicity, and adaptability inevitably arise in global computing frameworks based on wide-area networks, where separately developed, autonomous applications need to collaborate by dynamical assembly.

An emblematic example is the service-oriented computing (SOC) paradigm, where service abstractions are published in public repositories, which can be queried by other services for discovering convenient partners to interact with. Since it is desirable that interactions are carried out in a safe way for all participants, different notions of contracts have emerged to assign responsibilities to each participant in a non-ambiguous way and to give suitable run-time guarantees whenever all participants respect the contract. Formally, this involves answers to two main question: 1) Are the service abstractions compatible with the contract? 2) Is each participant consistent with its abstract description? Note that here we are often concerned with the operational behaviour of services, not just with functional aspects. Moreover, the notions of compatibility and consistency should be such that whenever 1) and 2) are answered in the affirmative, then the overall interaction is guaranteed to be sound (e.g. absence of deadlocks, or type-safe communications, or absence of livelock under fair scheduling assumptions). While traditionally all checks are performed statically, the challenge posed by SOC is moving them to run-time with dynamic assembly.

* Research supported by the EU within the FET-GC II Integrated Project IST-2005-016004 Sensoria and by the Italian FIRB Project Tocai.it.

We can single out three different phases occurring in every collaboration: 1) negotiation; 2) commit; 3) execution. In (1) the prospective participants negotiate some guarantees in order to define a sort of contract. In (2) each participant can either accept or reject the contract. If they accept, the contract will bind their behaviours in (3) to guarantee a globally correct execution. The scheme given by the phases (1-3) is called NCE and it covers a wide range of situations like transaction processing (phases 1-2), session-based interactions (phases 2-3), and applications of proof-carrying code (phases 1 and 3). Necessarily we need to allow part of the verification to be done also at run-time, on the basis of both statically and dynamically negotiable information.

In this paper we deal with multi-party interactions, where the participants can negotiate the interaction protocol to follow according to their abstract behaviour. The outcome of the negotiation is a (possibly non-deterministic and concurrent) well-behaving, global contract that binds the admissible interactions, thus fixing exact responsibilities in case of faulted or misbehaving execution.

Our formalisation relies on Petri nets and it builds on classical results from concurrency theory and workflow management. In particular, it takes inspiration from Zero-Safe nets [1], unfolding construction [2, 3], and workflow nets [4]. Roughly, the procedure can be outlined as follows: each participant describes its admissible behaviour as a Zero-Safe net; the composition of all Zero-Safe nets is unfolded as a non-deterministic process; the unfolding is suitably pruned (e.g., by removing faulted situations) and presented as a contract K; if accepted by all participants, then their stable tokens are tagged according to K and the firing of their transitions apply local consistency checks on such tags. The main result establishes that any such execution cannot deadlock.

Synopsis. In § 2 we fix the notation and the key concepts needed in the rest of the paper. In § 3 we outline the various formal steps of our approach, grouping them according to the three-phases NCE classification. We deal with multi-party interactions, but due to space limitation, only two-parties (buyer and seller) are considered in our running example. Some concluding remarks are in § 4.

2 Preliminaries

Petri nets. Place/Transition Petri nets (PT nets) are bipartite graphs that represent some kind of concurrent automata. Formally, a PT *net* is a triple $N = (S, T, F)$ where S is the set of places, T is the set of transitions and $F : (S \times T) \cup (T \times S) \to \mathbb{N}$ is the flow relation. The states of a PT net, called *markings*, are multisets $u : S \to \mathbb{N}$ of places, representing the number of *tokens* in each place. We write a marking as the formal sum $u = \bigoplus_i n_i a_i$, i.e. as an element of the free commutative monoid $S^{\oplus}$ (monoidal composition is defined by $(\bigoplus_i n_i a_i) \oplus (\bigoplus_i m_i a_i) = \bigoplus_i (n_i + m_i) a_i$, with 0 as the unit). Multiset inclusion is written $u \subseteq v$ (if $u(a) \leq v(a)$ for all places a), and multiset difference $v \ominus u$ (it is defined only when $u \subseteq v$, with $(v \ominus u)(a) = v(a) - u(a)$ for all places a).

For any transition t, its *pre-* and *post-set*, written $\mathrm{pre}(t)$ and $\mathrm{post}(t)$ respectively, are the multisets over S such that $\mathrm{pre}(t)(a) = F(a, t)$ and $\mathrm{post}(t)(a) =$

$F(t,a)$, for all $a \in S$. We write $t : u \to v$ for a transition t with $\mathrm{pre}(t) = u$ and $\mathrm{post}(t) = v$ and say that t is *enabled* in the marking w if $\mathrm{pre}(t) \subseteq w$. If $t : u \to v$ is enabled in w, then t can be *fired* in w leading to $w' = w \ominus u \oplus v$ and its firing is written $w[t\rangle w'$. A *firing sequence* from w to w' is a sequence of firings $w_1[t_1\rangle w'_1$, ..., $w_n[t_n\rangle w'_n$ such that $w = w_1$, $w' = w'_n$ and $w_{i+1} = w'_i$ for all $i \in [1, n-1]$. A marking v is *reachable* from u if there is a firing sequence from u to v.

Unfolding. Starting from a net N and a marking u, the *reachability graph* $\mathcal{R}(N,u)$ is a graph whose nodes are all markings reachable from u in N and whose arcs are all triples $(u, t, u \ominus \mathrm{pre}(t) \oplus \mathrm{post}(t))$ such that $\mathrm{pre}(t) \subseteq u$. While the reachability graphs account for the interleaving description of the computational space of N, the interplay between non-determinism, causality and concurrency available in N is accounted for by the so-called *unfolding construction* $\mathcal{U}(N,u)$. Formally, $\mathcal{U}(N,u)$ is a non-deterministic occurrence net (i.e., an acyclic net, where transition pre- and post-markings are sets instead of multisets and where each place has at most one entering arc, i.e., backward conflicts are not allowed), together with a net homomorphism from $\mathcal{U}(N,u)$ to N that tells which places and transitions of the unfolding are instances of the same element of N. Roughly, the transitions of $\mathcal{U}(N,u)$, called *events*, represent all the possible firings of transitions in N in all possible runs of the net, and the places of $\mathcal{U}(N,u)$ are all the possible tokens that can be generated.

For occurrence nets, the notion of *causally dependent*, of *conflicting* and of *concurrent* elements can be represented by the binary relations $\preceq$, $\#$ and $\mathbf{co}(\_,\_)$, respectively. Formally, $\preceq$ is the transitive and reflexive closure of the *immediate precedence* relation $\prec_0 \stackrel{\text{def}}{=} \{(a,t) \mid a \in \mathrm{pre}(t)\} \cup \{(t,a) \mid a \in \mathrm{post}(t)\}$. Letting $t_1 \#_0 t_2 \stackrel{\text{def}}{\Leftrightarrow} t_1 \neq t_2 \ \wedge \ \mathrm{pre}(t_1) \cap \mathrm{pre}(t_2) \neq \emptyset$, binary conflict $\#$ is defined as the minimal symmetric relation that contains $\#_0$ and that is hereditary with respect to $\preceq$ (i.e., such that if $x_1, x_2, y \in S \cup T$ and $x_1 \# x_2$ and $x_1 \preceq y$ then $y \# x_2$). The concurrency relation is defined by $\mathbf{co}(x_1, x_2) \stackrel{\text{def}}{\Leftrightarrow} \neg(x_1 \prec x_2 \vee x_2 \prec x_1 \vee x_1 \# x_2)$ and it is extended to sets of elements by letting $\mathbf{co}(X) \stackrel{\text{def}}{\Leftrightarrow} \forall x_1, x_2 \in X \ \mathbf{co}(x_1, x_2)$.

The net $\mathcal{U}(N,u)$ is defined (up to iso) as the net generated by the rules in Table 1, whose places have the form $\langle a, H, k\rangle$ and whose transitions have the form $\langle t, H\rangle$, where $a \in S$, $t \in T$, H is a set of causes that encodes the history of the element, and k is a positive natural number used to distinguish different tokens with the same history. The top rule introduces a distinguished place (with empty history) for each of the tokens in the initial marking u. The bottom rule adds an event e that represents the firing of a transition that consumes the tokens in Θ. The condition $\mathbf{co}(\Theta)$ rules out all inapplicable firings. The event e introduces the elements in Υ, representing the tokens produced by the corresponding firing.

The elements in $\mathcal{U}(N,u)$ can be stratified according to the depth function $\delta : S_{\mathcal{U}(N,u)} \cup T_{\mathcal{U}(N,u)} \to \mathbb{N}$ defined as follows: for places we let $\delta(\langle a, \emptyset, k\rangle) = 0$ and $\delta(\langle a, \{e\}, k\rangle) = \delta(e)$, while for transitions we let $\delta(\langle t, H\rangle) = 1 + \delta(H)$, where $\delta(H) = \max\{\delta(x) \mid x \in H\}$. A set of elements Θ is a cut if $\mathbf{co}(\Theta)$ and there is no element x such that $\mathbf{co}(\Theta \cup \{x\})$; it is maximal if there is no $x \in \Theta$ and y not conflicting with elements in Θ such that $x \preceq y$.

$$\frac{u(a) = n,\ 1 \leq k \leq n}{\langle a, \emptyset, k\rangle \in S_{\mathcal{U}(N,u)}}$$

$$\frac{t : \bigoplus_{i\in I} a_i \to \bigoplus_j n_j b_j \in T_N, \quad \Theta = \{\langle a_i, H_i, k_i\rangle \mid i \in I\} \subseteq S_{\mathcal{U}(N,u)}, \quad \mathbf{co}(\Theta)}{e = \langle t, \Theta\rangle \in T_{\mathcal{U}(N,u)},\ \Upsilon = \{\langle b_j, \{e\}, k\rangle \mid j \in J,\ 1 \leq k \leq n_j\} \subseteq S_{\mathcal{U}(N,u)},\ \mathrm{pre}(e) = \Theta,\ \mathrm{post}(e) = \Upsilon}$$

Table 1. The unfolding $\mathcal{U}(N, u)$.

We shall use the unfolding to distill a global contract binding the interaction of participants at run-time.

Zero-safe nets. Zero-Safe nets (ZS nets) [1] are a transactional variation of PT nets, where the set of places is partitioned in two sets, of Zero-Safe places Z and stable places L, respectively. Stable markings (i.e., multisets of stable places), describe the observable states of the system, whereas the presence of tokens in Zero-Safe places denotes a marking as transient, i.e., internal to a transaction segment. The firing rule is the usual ones, except for the fact that all stable tokens produced during the transactions are made available only at the end of the transaction, when all zero-safe tokens have been consumed.

The corresponding firing rules can be explained as follow. Given a ZS net B, take the PT net $\hat{B}$ obtained from B by introducing primed version a' of all stable places a and by renaming the flow relation so to produce tokens in primed places instead of in the corresponding original stable places. More precisely, let $\hat{B}$ such that $S_{\hat{B}} = S_B \cup \{a' \mid a \in L_B\}$, $T_{\hat{B}} = T_B$ and $F_{\hat{B}}(x, y) = 0$ and $F_{\hat{B}}(x, y') = F_B(x, y)$ if $y \in L_B$, while $F_{\hat{B}}(x, y) = F_B(x, y)$ otherwise. Then a firing sequence of $\hat{B}$ is called a transaction fragment, a transaction is any firing sequence from a marking $u \in L_B^{\oplus}$ to $v \in (S_{\hat{B}} \setminus Z_B)^{\oplus}$, and its commit transforms all the primed tokens in v to ordinary tokens, i.e., leads from v to $\hat{v}$ such that $\hat{v}(x) = v(x) + v(x')$ for all $x \in L_B$ and $\hat{v}(z) = v(z)(= 0)$ otherwise.

We shall exploit ZS nets to model the interaction protocols of participants, where zero-safe places correspond to intermediate states and the protocol is implicitly concluded when only stable tokens are around.

Workflow nets. Similar ideas appear in workflow nets [4], that have a distinguished start place a_{start} with no incoming arc and a distinguished end place a_{end} with no exiting arc, playing the role of stable places in ZS nets. In fact, a workflow net is *weakly sound* if any computation that starts with a token in the start place can always terminate reaching the marking with a unique token in a_{end}. Again, all the places different from a_{start} and a_{end} are considered as modelling internal, intermediate states. We shall generalise the notion of sound workflow nets to the case where many start and end places are present and where the soundness if relative to a suitable notion of execution contract.

3 Open transactions

In this section we illustrate our approach according to the NCE scheme.

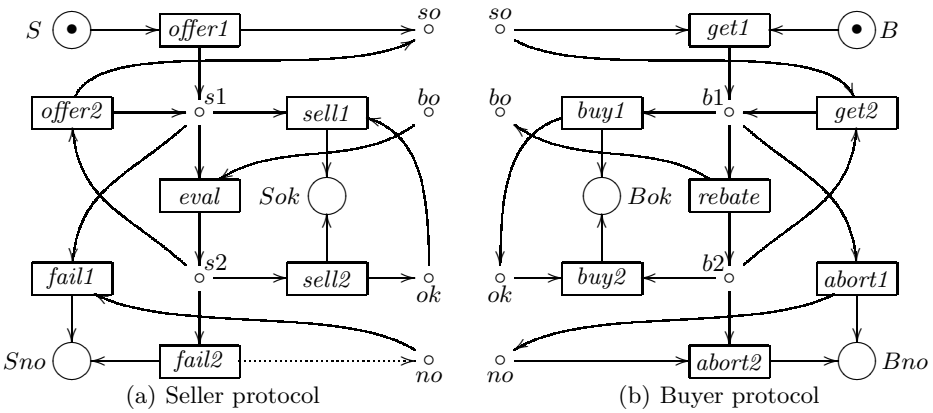


Fig. 1. Two local nets

3.1 NCE1: negotiation

The negotiation phase takes into account the available options of each participant to distill a global interaction protocol with behavioural guarantees for all.

Local and global Zero-Safe nets. The basic idea is to model interaction protocols as ZS nets: each participant describes its possible behaviour as a Zero-Safe net B_i, together with a stable marking u_i. We call these nets *local*. Each local net asserts that starting from the stable marking u_i a correct interaction should be guaranteed to lead to some other stable marking. Each local transaction is *open* in the sense that it may require some exchange of tokens with other local nets (typically, via zero-safe places only).

Given n local ZS nets such that their set of transitions are pairwise disjoint, a *global* ZS net $B = \bigcup_{i=1}^{n} B_i$ is then obtained as the union of all B_i's, with initial stable marking $u = \bigoplus_{i=1}^{n} u_i$. Typically, the local nets will share just certain zero-safe places, that are used to coordinate the local choices of participants.

Alike sound workflow nets, the ideal situation would be that starting from u any computation in $\hat{B}$ would eventually lead to a stable marking: this way we would be guaranteed that no matter which local choice is performed, no faulted situation can arise that leaves pending tokens in zero-safe places. Unfortunately, this is a too strong requirement, unrealistic in most situations, because local nets are developed according to different needs and separately from the others.

For example, consider a simple two-party situation, with a seller and a buyer whose local nets are in Fig. 1 (circles are places, smaller if zero-safe; boxes are transitions; arcs model the flow; bullets are tokens). Their interaction begins with an offer from the seller. Both the buyer and the seller may accept the last offer proposed by the other, or make a different offer, or abort the negotiation. The global net $S \cup B$ is well-behaving, according to the criteria explained above,

because each transaction can eventually lead either to the marking $Sok' \oplus Bok'$ or to the marking $Sno' \oplus Bno'$. On the other hand, the transaction might not terminate if the strategy of both parties is to make offers repeatedly.

Now suppose a different seller protocol is taken, where the local decision of abandoning the negotiation is not communicated to the buyer (i.e., the arc $(\mathit{fail2}, \mathit{no})$ is removed, whence the use of a dotted line in Fig. 1). Then the global net might present erroneous transaction segments, that cannot be completed, like the firing sequence *offer1*, *get1*, *rebate*, *eval*, *fail2* that leads to the marking $Sno' \oplus b2$, which is deadlock and not stable.

Contract unfolding. The unfolding of the global net $\hat{B}$ gives a complete view of the possible interactions that can take place. In particular, the above example illustrates two typical symptoms of problematic execution, easily generalised to: 1) unfolding is not finite, 2) the unfolding exposes non stable deadlocks. The solution we propose is to model contracts as partial unfoldings, i.e. finite non-deterministic processes that satisfy some additional constraints.

The first issue can be dealt with just by assuming some bound d on the depth of elements considered in the unfolding. The bound can be enforced by adding the condition $\delta(\Theta) \leq d$ to the second rule in Table 1. Another possibility is to specify a different depth bound d_t for each transition t, in which case the condition to be added in Table 1 is $\delta(\Theta) \leq d_t$. The bound d itself can emerge, e.g., as the conjunction of the bound required by each participant when declaring their interest in the negotiation, which amounts to some sort of time-out for carrying out the whole execution. We name this property *depth boundedness*.

The second issue can be dealt with by pruning those events that may stall the execution. Formally, this can be characterised by requiring that any maximal cut contains no zero-safe place. We name this property *stability*.

A *global contract net* K is a non-empty, depth-bound and stable subnet of $\mathcal{U}(\hat{B}, u)$, i.e. a particular non-deterministic process of $(\hat{B}, u)$.

Contract pruning. When stability is violated, we can of course apply some pruning to remove those elements that cause troubles. Let Θ be a maximal cut such that $x = \langle z, \{e\}, k\rangle \in \Theta$ for some zero-safe place z, then we remove e and all y such that $e \preceq y$ (thus also x is removed) from the already partial unfolding and then iterate the pruning on the result. Clearly, as the initially considered partial unfolding is finite and at least two elements are removed at each step of the pruning algorithm, then the pruning algorithm terminates. Moreover, its result enjoys stability by definition. However, the algorithm is not always confluent. Consider the net in Fig. 2. The only problematic cut is $\{z_1, z_3\}$: if we decide to remove z_1 and therefore e_1, then we get a global contract net and we are done; viceversa if we decide to remove z_3, then also e_3, e_4 and c are removed and the resulting net has two maximal cuts $\{z_1\}$ and $\{z_2\}$ that in turn require the removal of e_1 and e_2 and we are left with the idle contract.

The algorithm can be made confluent by removing at each single step all available candidates. We call this strategy *drastic pruning*. In the example above,

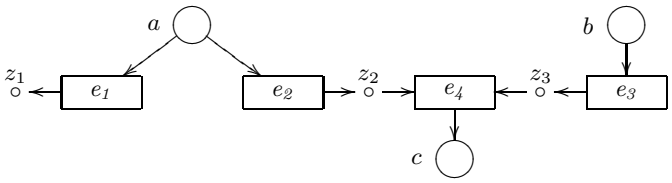


Fig. 2. A non-stable partial unfolding

e_1 and e_3 would have been removed at the first step (together with all their descendants), and e_2 at the second step, leaving again the empty contract.

Figure 3(a) shows the depth-bound unfolding of our running example (in the absence of arc $(\mathit{fail2}, \mathit{no})$), up to depth 7. It is not a global contract net, because it is not stable. In fact it has seven maximal cuts, but only three of them are stable: 1) $\{\mathit{Sok1}, \mathit{Bok1}\}$, 2) $\{\mathit{Sno1}, \mathit{Bno1}\}$, 3) $\{\mathit{Sok2}, \mathit{Bok2}\}$, 4) $\{\mathit{Sno2}, \mathit{b21}\}$, 5) $\{\mathit{s12}, \mathit{ok3}, \mathit{Bok3}\}$, 6) $\{\mathit{s12}, \mathit{no2}, \mathit{Bno2}\}$ and 7) $\{\mathit{s12}, \mathit{bo2}, \mathit{b22}\}$. The global contract net produced by drastic pruning algorithm is in Fig. 3(b).

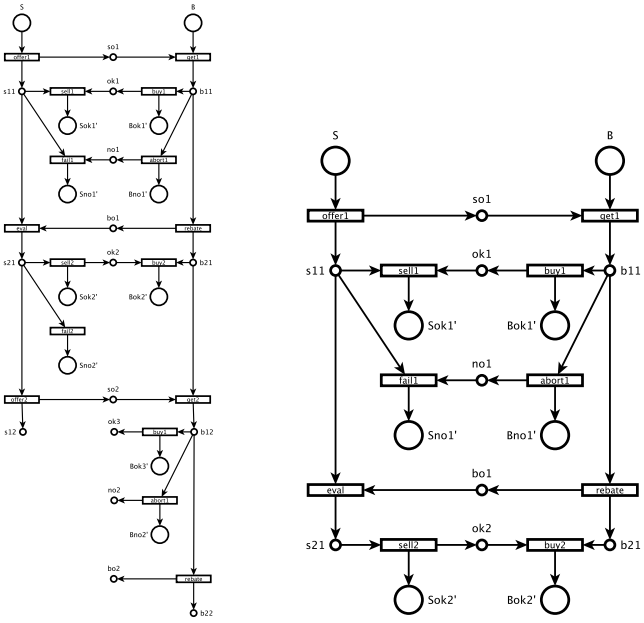
3.2 NCE2: Commit

The global contract net K can then be used as some sort of typing information that can be attached as a tag to the tokens and used in a prescriptive way during the execution phase. Hence, if the global contract net is considered viable from all participants, then it is attached as a decoration to each of the tokens in u, together with a fresh session identifier σ (the same for all tokens in u) and with the name of the token as given in K. In our running example, the two tokens in places S and B will thus be typed as $\langle \sigma, K, \langle S, \emptyset, 1\rangle\rangle$ and $\langle \sigma, K, \langle B, \emptyset, 1\rangle\rangle$, respectively. The typing information will be needed during the execution to guarantee that the contract will not be violated. We remark that in general it is not necessary to record the whole K in all tokens, but for each token x it would suffice to store the subnet of K consisting of all elements y such that $x \preceq y$.

Of course, if the global contract K is not acceptable for some participants, or if there are not enough stable tokens available, then the negotiation is considered aborted, and a new attempt has to be made. For example, the global contract net in Fig. 3(b) would constrain the seller to accept the possible rebate of the buyer, if any, which is disputable. Instead a global contract net that exclude the rebate would likely be a more convenient option.

3.3 NCE3: Execution

The final step of our approach consists in constraining the firing rules of the executable nets of all participants in order to inspect and respect the type information. Note that the executable net E_i of the ith participant can be larger than the local nets B_i exposed in the negotiation, i.e. it may contain other places and transitions and exhibit a higher degree of non-determinism without compromising the correct execution of the contract.



(a) Depth-bound (b) Global contract

Fig. 3. Building a global contract net

Tagged firing. Let E be the executable ZS net of a participant and suppose that a transition $t : \bigoplus_{i=1}^{n} a_i \to \bigoplus_{j \in J} n_j b_j \in E$ (possibly involving zero-safe places) is currently enabled and ready to consume the tagged tokens $a_1 : \tau_1, ..., a_n : \tau_n$, where $\tau_i = \langle \sigma_i, K_i, x_i \rangle$ is the type information attached to the ith token. Then t can fire if: 1) for any pair of indexes $i, i' \in [1, n]$ we have that $\sigma_i = \sigma_{i'}$; 2) each K_i contains an event $e = \langle t, \Theta \rangle$, where $\Theta = \{x_i \mid i \in [1, n]\}$. Without loss of generality, we denote by σ and K the common session identifier and global contract net of the above tokens. The firing of t will then produce the multiset of stable, non-decorated tokens $\Upsilon' = \{n_j b_j \mid j \in J, b_j \text{ is stable}\}$ and the set of tagged tokens $\Upsilon = \{b_j : \langle \sigma, K, \langle b_j, \{e\}, k \rangle \rangle \mid j \in J, b_j \text{ is zero-safe}, 1 \leq k \leq n_j\}$. Note that stable tokens are released non atomically by E, but this is not important, because we are guaranteed that the transaction will end in a finite amount of time and with no zero-safe token left. In fact, conly the choices that have been accounted for in K can be realised, and the properties of K guarantee that the overall execution is sound, no matter which local choices are taken by each participant. Finally, we remark that not all events in K must take place, because in general K can be a non-deterministic net.

4 Concluding remarks

We have proposed a net-based model for multi-party open transactions developed according to the NCE scheme. Its main features are the dynamic stipulation of contracts, the guaranteed execution and the original mix of session-based interactions and transactional distributed activities. The model is based entirely on Zero-Safe nets and the negotiated global contract is a particular non-deterministic process that can be automatically distilled from the local protocol specifications of participants. This is better suited than, say, a deterministic process, because run-time choices are unavoidable in open transactions. It is also preferable to a soundness check over the conjunction of protocols, because the check would impose too strong compatibility requirements (deemed to fail in most cases) and would not guarantee termination (and consequently would require a distributed commit phase after the execution of the transaction).

A problem that remains open is to find suitable metrics for evaluating and comparing pruning strategies different from the (confluent) drastic algorithm assumed here, which can be unnecessarily restrictive in many cases. It also remains to be investigated how our approach can be amalgamated with the recent strand of proposals that exploit workflow nets or process calculi to model various kind of compatibility notions and global / local contracts: due to space limitation we give here just a few pointers to related work for the interested reader [4–11].

References

1. Bruni, R., Montanari, U.: Zero-safe nets: Comparing the collective and individual token approaches. Inf. Comput. **156**(1-2) (2000) 46–89
2. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part i. Theor. Comput. Sci. **13** (1981) 85–108
3. Meseguer, J., Montanari, U., Sassone, V.: Process versus unfolding semantics for place/transition petri nets. Theor. Comput. Sci. **153**(1&2) (1996) 171–210
4. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. BPM'00. Vol. 1806 of LNCS, Springer (2000) 161–183
5. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. ESOP'07. Vol. 4421 of LNCS, Springer (2007) 2–17
6. Laneve, C., Padovani, L.: The pairing of contracts and session types. Concurrency, Graphs and Models. Vol. 5065 of LNCS, Springer (2008) 681–700
7. Bruni, R., Mezzina, L.G.: Types and deadlock freedom in a calculus of services, sessions and pipelines. AMAST'08. Vol. 5140 of LNCS, Springer (2008) 100–115
8. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. CONCUR'08. Vol. 5201 of LNCS, Springer (2008) 418–433
9. Bravetti, M., Zavattaro, G.: Contract-based discovery and composition of web services. SFM'09. Vol. 5569 of LNCS, Springer (2009) 261–295
10. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5) (2009)
11. Vasconcelos, V.T.: Fundamentals of session types. SFM'09. Vol. 5569 of LNCS, Springer (2009) 158–186

From Orchestration to Choreography: Memoryless and Distributed Orchestrators

Sophie Quinton, Imene Ben-Hafaiedh and Susanne Graf

Université Joseph Fourier / VERIMAG,
Grenoble, France
{quinton, benhfaie, graf}@imag.fr

Abstract. In the context of Web services, making client and service interact so as to satisfy the client, that is, making the service *compliant* with the client, can be done using either orchestration or choreography. In this paper we propose to build, whenever possible, memoryless orchestrators, and then distribute them using protocols so as to obtain choreographies.
When necessary for guaranteeing compliance, we infer from the initial (sequential) transition system possible concurrency between certain interactions, whose validity must be checked by the designer. Our approach allows for a clear distinction between the design phase and the implementation phase while being, in the general case, more efficient than orchestration. An example dealing with resource management illustrates the usefulness of memoryless orchestrators. We also discuss a methodology allowing contract-based design and verification of Web services at a higher level of abstraction.

1 Introduction

When designing Web services, important issues are those of *compliance* with a set of potential clients, and preservation of compliance by *refinement* (see [1], [2]). Indeed, in order to prove compliance for a large class of clients and services based on the use of abstractions of real clients and services, it is crucial to determine a notion of refinement for clients and for services (potentially different).

Here we focus on another issue specific to Web services, namely, deciding whether the *service adaptation* for a given class of clients and services should be realized by an *orchestration* or a *choreography*.

An orchestrator is a global mediator between the service and client components. Orchestrators can be automatically synthesized for given abstract definitions of client and service, see e.g. [2]. Their main drawback is that they are centralized, that is, all interactions between service and client components are controlled by the orchestrator. Very often, this is not needed. Let us consider, e.g., n clients sharing k resources. It is possible to predefine for each client a preferred "local" resource which can be obtained through local negotiation. Thus only in the rare cases where the preferred resource is not available would a more global communication be needed.

Choreographies are distributed controllers and avoid that problem. The problem there lies in their implementation. In [3], e.g., *realizability* of choreographies is studied and can be enforced by extra communications. However, the expressivity of the given specifications is limited.

Note that in general, distribution of an orchestrator leads to a very inefficient choreography. The reason is that orchestrators tend to be over-constrained with respect to message orderings. As a result, large numbers of messages are needed to provide every component with the required knowledge about the global state to decide about the next global interaction to be taken.

More simply, we restrict the class of orchestrators which we try to distribute to those definable by a memoryless orchestrator represented by a priority order and a set of local bounded buffers. The BIP framework [4,5] is a natural candidate for this purpose as it allows naturally representing clients and services as component behaviours, their interactions as the interaction layer and memoryless orchestrators as the priority layer (see section 2).

The priority rules representing orchestrators can be computed in two phases: the first one focuses on violations of compliance that can be resolved by adding finite (reordering) buffers. The second phase infers, if possible, a set of static priorities between interactions making the remaining deadlock states unreachable.

Properties expected from Web services are analyzed on this generated BIP model, which is still not too abstract and for which efficient verification methods exist (e.g. structural ones [6], [7][1]). The synthesized

[1] Those approaches do not require the computation of the global state space, although we construct it anyway so far to generate the memoryless orchestration.

memoryless orchestrators can then be implemented by means of protocols communicating amongst each other as locally as possible, as determined by static analysis. When there are no (or few) interactions dominated by a priority, that is, if the given client and service were initially (almost) compliant — up to some reordering being achieved by local buffers, the implemented system will be as efficient as a rendez-vous based choreography (e.g. [3]). When components have in many states a deterministic interaction set, as little as a two-way message exchange between the two peers may then suffice to realize an interaction (see Section 4).

On the other hand, there are cases in which the implemented system will be less efficient than a memoryful orchestrator. This may typically happen if the depth of the priority order is large, leading to global interactions. The other key issue with choreographies is the existence of global choices. In this respect, our algorithm is rather efficient, at least with respect to the criterion we have chosen here: make progress as quickly and as locally as possible, also at the cost of additional, potentially useless, message exchanges. Additional (static) analysis of specifications may allow decreasing the message overhead. For example, knowledge about persistency of readiness or enabledness of certain interactions avoids multiple requests for the same information, or allows choosing a preferred initiator for an interaction.

We believe that an interesting contribution of our work is the clear separation between the design phase of Web services, in which all the important properties are guaranteed by using our proposed synthesis algorithm and additional verification of any required safety or progress property, and the implementation phase which is expected to be fully automated. This distinction hopefully allows obtaining more abstract specifications and thus more efficient verification. We propose to go a step further by designing Web services using structured interactions as in BIP.

The paper is organized as follows: Section 2 shortly presents the BIP framework on which our intermediate representation of memoryless orchestrators relies. Section 3 explains how memoryless orchestrators are synthesized while Section 4 sketches how they are implemented by protocols. Finally, Section 5 discusses a possible design methodology integrating verification and implementation of Web services and discusses refinement issues.

2 The BIP framework

In [4,5,8], the framework BIP for component-based design and verification has been proposed. In BIP, systems are built by superposing three layers of modeling: Behavior, Interaction, and Priority. The classic notion of input/output of synchronous frameworks is replaced by the more expressive notion of multi-party interaction which allows each of the involved interaction partners to impose constraints on when the interaction may take place and imposes no notion of input completeness.

BIP is related to process algebras such as CCS [9] or CSP [10] by its rendez-vous-like interaction mechanism (on a set of *ports*) and the restriction to a strictly local notion of state. It has been shown however in [11] that the set of composition operators that can be defined within the BIP framework is, according to a definition taking into account the ability to coordinate components, more expressive than composition operators of CCS, CSP and SCCS [12]. BIP also addresses the problem of composition of operators and of their properties, which can be exploited for structural verification [8].

For the sake of clarity, we present here a simplified version of the BIP framework, without variables, guards or data transfer.

Definition 1 (Labeled Transition System). *A* Labeled Transition System *(LTS) is a tuple* $(Q, q^0, \mathcal{P}, \delta)$ *where* Q *is a set of states,* $q^0 \in Q$ *is an initial state,* $\mathcal{P}$ *is a set of labels and* $\delta \subseteq Q \times 2^{\mathcal{P}} \times Q$ *is a transition relation.*

LTS are used to represent behaviors of components. In this context, labels refer to the *ports* with which transitions are associated. Let us emphasize that transitions are labeled by sets of ports because a component or subsystem may be able (or even required) to interact through several ports simultaneously.

Components K interact via their ports. An *interaction* is characterized by the set of ports which synchronize, generally involving more than one component. A *connector* c is characterized by a set of ports and a set of interactions, describing all possible synchronizations involving the port set of c. Typical connectors represent rendez-vous or broadcast but also mutual exclusion, when only interactions involving a port p of a resource and the corresponding $\bar{p}$ of a single component are part of c.

Definition 2 (Interaction model). *A connector* c *is a set of interactions; we denote* $ports(c)$ *the set of ports that are involved in at least one of the interactions in* c*. An* interaction model *on a set of ports* $\mathcal{P}$ *is a set of connectors* $\mathcal{C}$*.*

Definition 3 (Legal interactions). *The set of* legal *interactions of an interaction model $\mathcal{C}$, denoted $\mathcal{L}(\mathcal{C})$, is $\{\bigcup_{l=1}^{k} i_l \mid k > 0 \wedge \forall l = 1..k, \exists c_l \in \mathcal{C}, i_l \in c_l,\ c_l$ pairwise distinct connectors $\}$.*

According to this definition, any combination of interactions of different connectors is a legal interaction. This notion encompasses concurrency, as interactions from different connectors may be fired simultaneously as well as not, unless stated otherwise by the component's LTS.

Definition 4 (Priority order). *A* priority order *on an interaction model $\mathcal{C}$ is a strict partial order such that:*

- $\forall c \in \mathcal{C}, \forall i, j \in c, i \subseteq j \implies i < j$
- $\forall i, j, \alpha \in \mathcal{L}(\mathcal{C}), i < j \wedge c(\alpha) \cap (c(i) \cup c(j)) = \emptyset \implies i \cup \alpha < j \cup \alpha$

where for any interaction i_1, $c(i_1)$ is the set of connectors that contain an interaction i_2 such that $i_2 \subseteq i_1$.

Priorities are used to arbitrate between simultaneously enabled interactions, for example to enforce scheduling policies. The first condition above ensures maximal progress between interactions of a single connector. The second condition ensures monotonicity, that is priorities are preserved from smaller to larger interactions.

Definition 5 (BIP system). *A system represented in BIP consists of a set of components K_i whose behaviors are given by LTS on disjoint port sets $\mathcal{P}_i$, an interaction model $\mathcal{C}$ on $\mathcal{P} = \bigsqcup_{i=1}^{n} \mathcal{P}_i$ and a priority order $<$ on $\mathcal{C}$.*

We refer to the three items thus defined as layers which are called respectively *behavior*, *interaction* and *priority*. We refer to $(\mathcal{C}, <)$ as a *composition operator* on $\mathcal{P}$, because interaction and priority express how to compose a set of LTS so as to make them interact. As usual, $q_1 \xrightarrow{\alpha} q_2$ denotes $(q_1, \alpha, q_2) \in \delta$ and $q_1 \xrightarrow{\alpha}$ denotes $\exists q' \in Q, q \xrightarrow{\alpha} q'$.

Definition 6 (Operational semantics). *Let $\{K_i\}_{i=1}^{n}$ be a set of LTS, where $K_i = (Q_i, q_i^0, \mathcal{P}_i, \delta_i)$. Let $\mathcal{P} = \bigsqcup_{i=1}^{n} \mathcal{P}_i$ and $\{\mathcal{C}, <\}$ a composition operator on $\mathcal{P}$. The composition of $\{K_i\}_{i=1}^{n}$ with $\{\mathcal{C}, \pi\}$ is an LTS $(Q, q^0, \mathcal{P}, \delta)$ such that $Q = \prod_{i=1}^{n} Q_i$, $q^0 = (q_1^0, \ldots, q_n^0)$ and δ is defined as follows.*
$\forall \alpha \in \mathcal{L}(\mathcal{C}), \forall q^1 = (q_1^1, \ldots, q_n^1), q^2 = (q_1^2, \ldots, q_n^2) \in Q, q^1 \xrightarrow{\alpha} q^2$ iff:

- $\forall i, q_i^1 \xrightarrow{\alpha_i} q_i^2$ *where $\alpha_i = \alpha \cap P_i$ and with the convention that $\forall q, q \xrightarrow{\emptyset} q$*
- $\nexists \alpha' \in \mathcal{L}(\mathcal{C}), \alpha < \alpha' \wedge q^1 \xrightarrow{\alpha'}$

Thus, only interactions that are locally enabled in all concerned components, and furthermore not inhibited by an interaction with higher priority, may be fired. Independent interactions may be fired jointly .

3 Synthesis of memoryless orchestrators

3.1 Contracts for Web services

Web services communicate via peer-to-peer exchange of messages. We assume that no message can be lost or reordered when sent through the same channel. An elegant way to represent message loss or absence of interaction peers in BIP is discussed in section 5. In this paper, we propose only time independent algorithms. In particular, we do not use on timeouts to guarantee progress. The framework could be extended to systems satisfying real-time requirements by also adding constraints on execution times and transmission delays.

In the context of Web services, *contracts* are used to describe how a client or a service is expected to behave. As in [2], we use a fragment of CCS to define such contracts. . We use as the input for our methods the LTS defined by a CCS term. In the following, x is a process variable, α denotes an action, and σ, ρ are process terms.

$$\sigma ::= 0 \mid \alpha.\sigma \mid \sigma + \sigma \mid \sigma \oplus \sigma \mid \mathtt{rec}\, x.\sigma \mid x$$

The terminated process is represented by 0, and $\alpha.\sigma$ represents sequence. As usual, $+$ denotes the *external* choice (whether the process $\sigma + \rho$ will behave as σ or ρ depends on the environment), $\oplus$ the *internal* (non-deterministic) choice. Variables are assumed to be guarded, that is, every free occurrence of x in a term $\mathtt{rec}\, x.\sigma$ appears in a subterm of the form $\alpha.\rho$. This ensures that no sequence of internal transitions is infinite. The process $\mathtt{rec}\, x.\sigma$ behaves like $\sigma\{^{\mathtt{rec}\, x.\sigma}/_x\}$, which is σ with every free occurrence of x replaced by $\mathtt{rec}\, x.\sigma$.

The set of actions appearing in σ is denoted $act(\sigma)$. The semantics of σ can be defined by an LTS $(Q, q^0, \mathcal{P}, \longrightarrow)$, where states are terms of σ: Q is the set of derivations of σ, $q^0 = \sigma$, $\mathcal{P} = act(\sigma) \cup \{\tau\}$ and $\longrightarrow$ is the transition relation given by the following SOS rules (symmetric rules for $+$ and $\oplus$ are omitted).

$$\alpha.\sigma \xrightarrow{\alpha} \sigma \qquad \sigma \oplus \rho \xrightarrow{\tau} \sigma \qquad \mathtt{rec}\, x.\sigma \xrightarrow{\tau} \sigma\{^{\mathtt{rec}\, x.\sigma}/_{x}\}$$

$$\frac{\sigma \xrightarrow{\tau} \sigma'}{\sigma + \rho \xrightarrow{\tau} \sigma' + \rho} \qquad \frac{\sigma \xrightarrow{\alpha} \sigma'}{\sigma + \rho \xrightarrow{\alpha} \sigma'}$$

Actions can be either outputs (denoted $\bar{a}$) or inputs (denoted simply a). By $\sigma \parallel \rho$ we denote the composition of σ and ρ where inputs and corresponding outputs must synchronize, which in BIP can be represented by a set of connectors of the form $\{\{\bar{a}, a\}\}$. In the sequel, when no ambiguity is possible, we refer to the connector $\{\{\bar{a}, a\}\}$ or to the interaction $\{\bar{a}, a\}$ as a.

Satisfaction of a client is represented by a special action e. A client is satisfied if it offers e. To check whether a given client and a given service can interact so as to satisfy the client, we need to define *compliance*.

Definition 7 (Compliance). *A service ρ is* compliant *with a client σ, denoted $\rho \vdash \sigma$, iff $\sigma \parallel \rho$ has no deadlock.*

By *deadlock*, we mean a state in which the client is not satisfied and no transition can be fired. A service τ *refines* a service ρ iff every client satisfied by ρ is also satisfied by τ. Based on these definitions, a service $a + b$ does not refine a, because the client $\bar{a}.e + \bar{b}.\bar{c}.e$ would be satisfied by the latter but not by the former. To avoid this, a looser notion of compliance, called *weak compliance*, can be used (see [2]). A service ρ is *weakly compliant* with a client σ if there exists an orchestrator which may interfere with the execution of ρ and σ as to satisfy the client, by storing outputs until a corresponding input is enabled, and, in the situation where both client and service offer an external choice, not choosing any interaction that may lead to deadlock.

We are interested in building light-weight memoryless orchestrators which can be expressed by a priority preorder over the set of interactions.

More precisely, given a service σ and a client ρ, we proceed in two steps.

Step 1: We infer concurrency. While building the synchronized product $\sigma \parallel \rho$, violations of the compliance relation are detected. If they can be avoided by reordering of messages that can be implemented using bounded buffers, then they are collapsed in the specification. Otherwise the transitions leading to deadlock are marked as *error*. Transitions that have been (in the view of the designer) erroneously collapsed can be marked as *error* as well.

Step 2: We infer priorities. If possible, a set of priorities is computed so as to make the erroneous states unreachable according to the operational semantics given in Section 2, by inhibiting some transitions. Static priorities are a nice trade-off between the need for some non-local dependencies and the risk to build a specification that would require a global state.

3.2 Inferring concurrency

Checking compliance between a client σ and a server ρ means checking of deadlock freedom of $\sigma \parallel \rho$. Let us consider the following client σ and service ρ.

$$\text{Client} : \bar{a}.b.c.e \qquad\qquad \text{Service} : a.\bar{c}.\bar{b}$$

Clearly, σ and ρ are not compliant, which is a bit of a pity because, a priori, the service does not need to interact with the environment between the outputs $\bar{c}$ and $\bar{b}$, and the order in which these outputs are presented to the client has no influence on the service[2]. In order to make the service and client compatible, we cannot modify the service itself. In [2] it is proposed to systematically add buffers between the component and the environment to allow the environment to pick interactions which can be made available in the desired order. In presence of choices, however, adding possible behaviors may also add new deadlocks occurring a few steps later. These deadlocks must then be eliminated by adding priorities. Therefore, we only add order inverting buffers when this allows eliminating a deadlock.

Definition 8 (Reordering of concurrent transitions). *Two transitions $t_1 = (q_1, a, q_1')$ and $t_2 = (q_2, b, q_2')$ of an LTS K are* concurrent *if $q_1' = q_2$ and a is an output, and furthermore there exists no external choice in conflict with a.*

[2] At least this is the case if the abstraction is precise enough to explicit all communications including an input.

When a is an output, it is possible to withhold the actual interaction a until after the execution of the interaction b because the behavior of K does not depend on the effect of K. If the interaction a is accepted by the environment after the interaction b but not before b, this allows avoiding a deadlock. If there exists an alternative to the interaction a, the future of K in q_1 *does* depend on the effect of a: speculating on b before a is indeed accepted by the environment may lead to a state outside the specification if an alternative for a ends up to be chosen. This is the reason why we propose such reorderings only if q_1 is a deadlock state, and no alternative is available.

Definition 9 (Trace sets, matching pairs). *As usually, we define the set of traces as the set obtained by reordering of concurrent interactions, but only from the left to the right. When $s_1.t_1.t_2.s_2 \in traces(s)$ and $t_1.t_2$ is concurrent, then $s_1.t_2.t_1.s_2 \in traces(s)$*

Two sequences s_1 and s_2 can be matched *if there exists $s'_i \in traces(s_i)$ such that s'_2 is obtained from s'_1 by inverting inputs and outputs.*

Instead of presenting the full algorithm, we illustrate it on the main situations to be handled. Let us first consider the simple Service above: while building the synchronized product of Client and Service, deadlocks are detected, here $b.\,c.\,e \parallel \bar{c}.\,\bar{b}$.

In this state, the Client can only interact on b, whereas the Service only on $\bar{c}$; the Service is deterministic and the smallest sequence of deterministic interactions containing both c and b — the only offer of the Client — is $s_1 = \bar{c}.\,\bar{b}$. Symmetrically on the Client side, we find $s_2 = b.\,c$. We find $traces(s_1) = \{\bar{c}.\,\bar{b}, \bar{b}.\,\bar{c}\}$ and $traces(s_2) = \{b.\,c\}$ and a matching pair $(\bar{b}.\,\bar{c}, b.\,c)$. A buffer realizing the required reordering in the Client, eliminates the deadlock.

In the case where s_1 and s_2 do not contain the same set of interactions, then if they can still be made matching by adding continuation, we can redo the same step until we know that the sequences can never match or we find a matching pair. In practice, it is not very reasonable to consider long sequences which in fact have to be remembered as such and are more a "patch" than a general solution. Also, if either the client or the server is distributed, they ressemble very much a memoryful orchestration, which we wanted to avoid. This is also the reason why we do not handle loops here, even if in principle this is doable.

If in the deadlock state several interactions are offered by the client or the service, any pair of interactions may be checked for a possible matching sequence pair, and we will retain only one of them.

If no multiset as described exists, or if the designer thinks that the concurrency inferred is erroneous, the corresponding states are marked as deadlocks. Otherwise, the size of buffer required for reordering is bounded by the largest number of actions collapsed on a single transition in the product.

3.3 Finding a sufficient priority order

Besides reordering, the orchestrator can also restrict external choice so as to avoid deadlocks. Indeed, if both client and service offer two actions as an external choice, then the orchestrator may choose which one is taken. Instead of presenting the full algorithm, we show how it works in several cases.

A simple example that works with priorities. An orchestrator can be represented with static priorities in the following example:

$$\text{Client} : \bar{a}.\,c.\,e + \bar{b}.\,d.\,e \qquad\qquad \text{Service} : a.\,\bar{d} + b.\,\bar{d}$$

The priority $a < b$ is sufficient to make the deadlock unreachable.

A simple example that does not work with priorities. Here is an example where an orchestrator cannot be represented with static priorities.

$$\text{Client} : \bar{a}.\,c.\,e + \bar{b}.\,\bar{b}.\,c.\,e + \bar{b}.\,\bar{a}.\,d.\,e \qquad\qquad \text{Service} : a.\,\bar{d} + b.\,b.\,\bar{d} + b.\,a.\,\bar{d}$$

Priorities cannot account for that, because avoiding deadlock both after a and after $b.\,b$ would require at the same time $a < b$ and $b < a$, which is impossible.

A short look at these examples makes it clear that weak compliance enforced via static priorities will not be preserved by refinement unless the priorities have a semantic meaning – which is not the case in our first trivial examples. Here is a detailed illustration of how the algorithm proceeds on the second example.

1. The product of client and service is computed. Deadlocks are detected and transitions leading to them are marked as *error*, as shown in figure 1.

2. State 1 of $\sigma \parallel \rho$ is the initial state and in this state b must be preferred to a in order to prevent a deadlock, so we conclude that any possible set of priorities making σ and ρ compliant includes $a < b$.
3. Besides, there are two possibilities to avoid the deadlock occurring if b is fired in state 2: either b has lower priority than a or state 2 is not reachable. The first option would lead to a contradiction, namely, $a < b$ and $b < a$ at the same time. The second option would imply that the transition from 1 to 2 labeled by b should be inhibited by a transition with higher priority enabled in 1. Such a transition can only be $1 \xrightarrow{a} 2$, which is also impossible. Thus we conclude that no memoryless orchestrator can make the given client and service compliant.

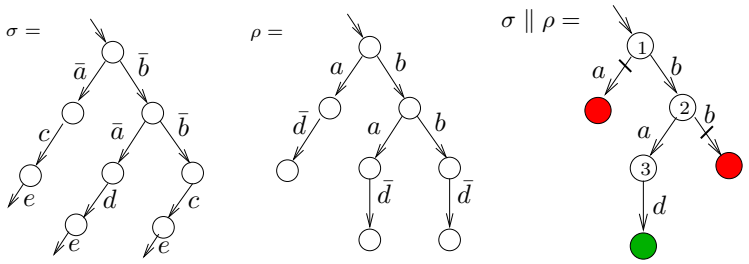


Fig. 1. A client σ, a service ρ and their product $\sigma \parallel \rho$.

The general algorithm is slightly more complicated because for each transition *error* one has to consider either adding a priority or removing a state.

If the algorithm fails to produce a priority order making the deadlocks unreachable, other approaches are possible. One is to infer more concurrency. We have inferred concurrency only in deadlock states. Some sequences of interactions that have been rejected might be made possible by adding priorities and thus satisfy the client.

It is important to underline the fact that the product of the client and the service is expected to be of a size comparable with both. As a matter of fact, it is even expected to be smaller because composition is likely to add constraints to both client and service. If there are several clients or services to be considered, then compositional approaches are possible. This will be discussed in section 5.

The dining philosophers example. Let us consider the dining philosophers example. The variant presented here is inspired from [2]. Philosophers are services who provide thought if they are given two forks by the resource. We consider here two philosophers and a resource with two forks. As usual, the deadlock arises if both philosophers are allowed to get one fork and never return them.

To handle this problem, always giving the highest priority to the request that is the closest to completion is a classic method for managing resources. Memoryless orchestrators are powerful enough to enforce such a policy. In the context of the dining philosophers, we thus give a different label for the action of getting the first or the second fork. The priority order then inferred $\{fork_1^\alpha < fork_2^\beta,\ fork_1^\beta < fork_2^\alpha\}$ is sufficient to prevent the deadlocks. For readability reasons, in figure 2, instead of interactions, we use the names of the ports that distinguish them and we give the same name $fork$ to 4 different ports of $Forks$ as they cannot be enabled at the same time.

Forks $= \mathtt{rec}\, x.\overline{fork}.\overline{fork}.thought.return.return.x$
Philo $= \mathtt{rec}\, x.fork_1.fork_2.\overline{thought}.\overline{return}.\overline{return}.x$

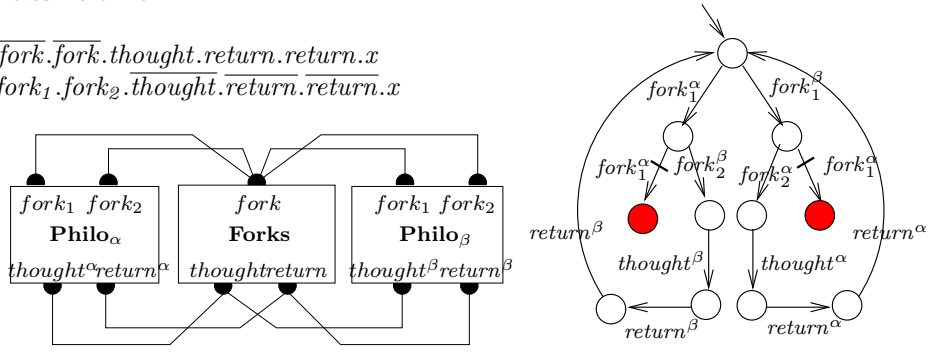


Fig. 2. A solution to the dining philosophers problem.

4 Implementation of memoryless orchestrators

BIP is defined by global semantics useful for ensuring progress properties with concise component specifications and proving them correct structurally [6]. Thus it is implemented by a global simulation engine [5] interacting with components.

In the context of web-services, we need a distributed engine. A partially concurrent implementation of BIP with message passing is given in [13], but still based on the existence of a centralized engine. Other proposals consist in adding a component for handling individual connectors, but this is not sufficient in presence of non prioritized choice or global priorities.

We propose a truly distributed BIP implementation by means of protocol agents associated with each component and which negotiate the next interaction to be executed as locally as possible. We extend existing algorithms for distributing process algebras such as [14] by handling global priorities. The protocol agents are machines communicating by exchanging messages via non lossy channels preserving the order of messages with the same source and destination; protocol read messages from their local message buffer by also preserving the order for any given message source.

Another specificity is that we rely on the BIP semantics of the system to satisfy properties, including deadlock freedom and fairness, required to guarantee client satisfaction. And we also want to consider distributed choice amongst a set of conflicting interactions without using static priorities. Besides, we assume components' internal activities to be terminating.

To define a correctness criterion for our algorithm, we provide two properties characterizing LTS obtained by composition using a BIP composition operator.

Notation 1 (Ready/enabled/conflicting/independent interaction) *An interaction c is* ready *in a (global) state q iff each port in c is locally ready. c is* enabled *in q iff c is ready in q and no interaction with higher priority is ready in q. Readiness, unlike enabledness, does not take priorities into account.*

As usual, two interactions are called conflicting *in state q if both are enabled but only one of them can be chosen in q. If both can be chosen jointly (according to the BIP multi-shot semantics) they are called* independent.

Property 1. Suppose a BIP system S defined as the composition of a set of components $\{K_i\}_{i=1}^n$ with a composition operator $(\mathcal{C}, <)$.

Safety. If in a state $q = (q_1, \dots, q_n)$ of S two interactions without a common component are *enabled*, then they must be independent. That is, all the states obtained by firing any of a, b or $a|b$ are reachable, and moreover, after the execution of $a|b$, or a and b in any order the same state is reached.

Progress. Any execution of S inevitably leaves q by executing at least one of the successors of S, if S has at least one successor in q. Any execution of S starting in q inevitably leaves q_i if there is no loop nor deadlock state containing q_i in S[3].

4.1 The distributed algorithm

For the sake of readability, we present an algorithm handling only binary rendez-vous connectors . We use c to denote this unique interaction as well as, when no mistake is possible, the corresponding local (input and output) ports. We do not consider true multi-shot semantics where a component is allowed to *require* interactions to occur jointly. See section 5 for a discussion about this issue.

Notation 2 *We call $ready_K(q_i)$ the set of* ports *enabled locally in a component K in state q_i and refer to it as the ready-set of K in state q_i.*

We denote by $k(c)$ the two components involved in a connector c. Given K and one of its connectors c, we write K_c for K's peer for c.

We denote by $dp(c) = k(\{b \mid c < b\})$ the set of components having an interaction with higher priority than c.

Our algorithm uses for deciding enabledness of c direct "negotiations" amongst components in $k(c)$ – which determine readiness – and then between one component in $k(c)$ (a "priority negotiator" chosen statically), and one component for each $a \in dp(c)$ (also a statically chosen negotiator). The algorithm is implemented by a protocol amongst protocol agents Pr_i. Each Pr_i is associated with a corresponding component K_i, which itself is obtained by enriching the original component by the set of buffers computed in the algorithm of section 3.2. The algorithm consists of the following main phases:

[3] we consider finite state LTS S.

1. a *busy* phase, in which an interaction to be executed has been chosen and the communication part is terminated. K_i is now executing the action part and the local ready-set in the successor state is not known yet. In this phase the protocol agent is quiet and defers all enquiries of other protocol agents by letting messages accumulate in its buffer[4]. Whenever the *busy* phase is entered or left all pending enquiries are handled. We suppose that no agent remains indefinitely *busy*.
2. a *ready* phase, in which the current ready-set $readySet$ has been communicated by K_i. Pr_i remains in this state until an enabled interaction has been found.
   To find an enabled interaction amongst $readySet$, the protocol checks $readySet$ in decreasing order of priority $<$. Determining if an interaction c is enabled by communication with its peer Pr_c and with $dp(c)$ is relatively straightforward (see algorithms in the annex).
   As soon as a c is found enabled, or if $readySet$ contains only one potential successor, Pr_i (tries to) execute c and sends a $COMMIT$ message to its peer[5].
   The interaction c is locally "done", if a $COMMIT$ has been sent to and received from Pr_i's peer,in any order. For simplicity, we suppose data to be piggybacked on the $COMMIT$ message[6].
3. whenever Pr_i is (from its own point of view) the first one to choose interaction a (and to send $COMMIT$), then it enters the *commit* phase, in which it waits for its decision to be committed — in which case the interaction is done and P_i provides the choice and the data to K_i and becomes *busy*. Or it is rejected, by an explicit $REFUSE$ which brings Pr_i back to *ready*. Any "counter proposal" that is received at this stage is $REFUSE$d, unless static analysis has detected the existence of a potential "decision cycle" in which case, there will be one statically determined *cycle breaker* who gets his choice done, and if Pr_i receives a counter proposal a from the *cycle breaker* — which may depend on the pair a, c — it will wait its own c to be accepted (no actual cycle exists) or refused (a cycle may exist), and only then send a response for a. Once, c or a is commited from both sides, all interactions for which a readyness request has been made or responded to, are explicitly $REFUSE$d.

Note that, when all protocols Pr_i are *ready*, the global state should be a state of S. Such a situation may never occur globally, and this is not needed. Whenever an interaction a is chosen, it is sufficient that all components which determine the enabledness of a in q — that is also those which may compromise a through a priority rule — are *ready* and in a state q compatible with S; the state of all other components is irrelevant.

5 Discussion

Memoryless orchestrators are not as powerful as, e.g., the *simple* orchestrators introduced in [2] in the sense that there are fewer pairs client/service that they make compliant. Besides, the complexity of our algorithm is even larger than the one for generating orchestrators. This is the price for having orchestrators from which we can automatically derive efficient distributed algorithms. Because concerns are clearly separated, our implementation does not have to take care of properties such as fairness. Obviously, the methodology presented here does not preserve compliance by refinement. However, using a notion of refinement under context as in [7] that takes into account priorities would allow reusability to some extent.

When designing Web services, it is possible and helpful to abstract away some implementation details. We propose to go a step further in this direction by using structured interaction, as is the case in BIP. The BIP framework described in Section 2 is much more powerful than the subset that has been used here. While using the priority layer we have kept interactions binary. Connectors can represent not only binary rendez-vous, but also n-ary rendez-vous or broadcast.

Instead of specifying group protocols in terms of sequences of binary interactions, we can express them at the specification level using, possibly a single, n-ary connectors, thus increasing the level of abstraction of the specifications. It is possible in BIP to represent loss of messages occurring when a component (process) receives a message that it cannot handle at the time of the reception. This is done by building a connector with two legal interactions, namely the output alone and the rendez-vous. Structural verification of BIP systems ([6]) could handle this efficiently, as well as multiple clients and services. In particular, the question of *realizability* (see [3]) can be decided very simply in BIP.

Designing Web services in a framework as expressive as BIP reinforces the distinction between design and implementation, allowing for a more abstract (and thus easier) design phase, more efficient verification,

[4] the size of the buffer can be statically bounded by the number of ports of K_i

[5] we aim at maximising progress, knowing that overall progress is enforced by the safety property of S.

[6] This is reasonable if the data volume is small or if conflicts are rare.

and automated implementation of the protocols realizing the specification. Expressing full BIP by protocols would require methods to decide when an orchestration component will be more efficient than protocols.

References

1. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Proc. of COORDINATION'07. Volume 4467 of LNCS. (2007) 96–112
2. Padovani, L.: Contract-directed synthesis of simple orchestrators. In: Proc. of CONCUR'08. Volume 5201 of LNCS. (2008) 131–146
3. Salaün, G., Bultan, T.: Realizability of choreographies using process algebra encodings. In: Proc. of IFM'09. Volume 5423 of LNCS. (2009) 167–182
4. Gößler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comput. Program. **55**(1-3) (2005) 161–183
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. of SEFM'06, IEEE Computer Society (2006) 3–12
6. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: Proc. of ATVA'08. Volume 5311 of LNCS. (2008) 64–79
7. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Proc. of SEFM'08, IEEE Computer Society (2008)
8. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. In: Proc. of EMSOFT'07, ACM Press (2007) 11–20
9. Milner, R.: A calculus of communication systems. In: LNCS 92. Springer (1980)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1984)
11. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: Proc. of CONCUR'08. Volume 5201 of LNCS. (2008) 508–522
12. Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. **25** (1983) 267–310
13. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Proceedings of FORTE'08. LNCS, Springer (June 2008)
14. Bagrodia, R.: Synchronization of asynchronous processes in CSP. ACM Trans. Program. Lang. Syst. **11**(4) (1989) 585–597

Inter-service Dependency in the Action System Formalism

Extended Abstract

Mats Neovius[1,2], Fredrik Degerlund[1,2], Kaisa Sere[1]

[1] Åbo Akademi University, Joukahaisenkatu 3 – 5, 20520 Turku, Finland
[2] Turku Center for Computer Science, Joukahaisenkatu 3 – 5, 20520 Turku, Finland
{mats.neovius, fredrik.degerlund, kaisa.sere}@abo.fi

1 Introduction

The use of formal methods is widely recognised for facilitating systematic construction of reliable and rigorous software. Methodologies supporting formalisation of functionality relying on distributed sources suggest to mastering inter-module dependencies where assignment of a global variable in one system changes the global state. The challenge comes to be identifying the properties and restrictions when formally defining dependencies involving the modules. The gain is that hence the systems need not to be administrated by a central entity and can be truly distributed. The modules can be independently replaceable as long as the functionality they guarantee remains intact given certain conditions. Consequently, the modules have well defined interfaces and they can be called services.

The motivation of our approach lies in that information sources become increasingly distributed; the information is provided by scattered independent entities [1, 2]. These entities depend on service(s) provide by other entities. A service can be an elementary source of information, some entity deducing new information depending on other (lower level) services or a combination of these two. Whether being an intermediate service or elementary source of information, the internal functionality of the provided service need not to be considered by the auxiliary services, i.e. the service can be considered a black-box.

The contribution of this extended abstract is in providing a glimpse into the research conducted in examining the means to formally rely on remote services and semantically reason about these. For this, we have defined an operator that masters the dependency relation that treats the remote services as stand-alone replaceable entities. Once mastering the formalisation of the services' interfaces, we claim that the formalism is ripe for specifying truly distributed inter-service dependent systems. We have chosen to model the dependency in the action system formalism framework, and we use reactive action systems as they provide means for reasoning about the information in a modular, distributed, manner. This extended abstract builds on our earlier work [3, 4].

2 Definitions of Concepts

For the reader to thoroughly understand this paper, some concepts need to be defined. We will consider systems that are either sources and/or utilisers of information. This paper will use *source* when indicating the origin of some information, realistically this is the input to the system or a sensor introducing some context. The *utiliser* constitutes the user of any provided information. Thus, an utiliser can be a source as well whenever it utilises other sources but changes these according to some rules such as calculating the mean value or by source introduction of its own. Realistically, this happens when a service depends on subservices to provide.

Because the source must not dictate its utiliser(s) but the utiliser selects the source(s), unidirectional dependencies are evident. Bidirectional dependencies in distinct traces are expressible, e.g. mutual agreement considering entities A and B where ↪ denotes "*depends on*"; A ↪ B and B ↪ A. We will also model *direct* and *indirect* dependencies. In direct dependencies the utiliser will halt until the source provides its service whilst in indirect the utiliser settles for being guaranteed that the source will eventually execute the task. In addition, we will use other concepts specific to the formalism that are introduced gradually.

3. Characteristics of an action

One way of formally modelling software is to focus on the *state space* of a program. Each state in the state space is identified by the disjoint conditions that hold in it. Changes in these conditions are of central interest and are traced. Because the current state is well defined as are the executable tasks, a weakest precondition predicate can be derived for each task. Deriving one predicate from another one coins the idea of a predicate transformer, originally introduced by Dijkstra.

3.1. Actions at a glimpse

Since providing a table listing all possible preconditions for all post-execution states of an action is unmanageable, due to its sheer size, the approach taken is to describe this as a function describing the weakest precondition of an action [5]. The action system framework is a state based formalism for defining distributed systems [6, 7]. The basic component in an action system is the *action*. It bases on Dijkstra's language of guarded commands [5, 8] and is defined with the *weakest precondition* predicate transformer, in short wp. From wp(A, q) we can derive the weakest precondition, i.e. the conditions for which executing action A the postcondition q is satisfied. These pre- and postconditions are mere predicates over state variables. The weakest precondition is defined for various actions as follows:

wp (*magic*, q)	$= true$	*Miraculous action*	(1)
wp (*abort*, q)	$= false$	*Aborting action*	(2)
wp (*skip*, q)	$= q$	*Stuttering action*	(3)
wp ($x \coloneqq E$, q)	$= q[E/x]$	*Multiple assignment*	(4)

wp (A; B, q)	= wp (A, wp (B, q))	*Sequential composition*	(5)
wp (A [] B, q)	= wp (A, q) ∧ wp (B, q)	*Nondeterministic choice*	(6)
wp ([a], q)	= $a \Rightarrow q$	*Assumption*	(7)
wp ({a}, q)	= $a \wedge q$	*Assertion*	(8)

The action *abort* is used to model disallowed behaviour, thus q is never satisfied, i.e. the outcome is *false*. *skip* is a stuttering action, not doing anything, thus, the weakest pre-condition for establishing post-condition q is q. $x := E$ is multiple assignment where all occurrences of x are substituted with an element in E, A; B is the sequential composition of two actions and A [] B the (demonic) nondeterministic choice between actions A and B. [a] is the assumption that is assumed true and {a} is called the assertion that is a predicate needed to evaluate true in order for the execution to proceed to guarantee q. For assumption, if 'a' is *false*, the action behaves magically whilst for assertion, if 'a' evaluates false, the action aborts. Hence the actions *abort* and *magic* can be seen as special cases of assertion and assumption, respectively.

An action A is enabled ($gd\ A$) whenever executing it does not establish an unwanted post-condition.

$gd\ A$	= ¬wp (A, *false*)	*Enabledness*	(9)

Hence, actions *abort*, *skip* and $x := E$ are always enabled.

This language allows guarded commands [gA]; sA, for convenience written $gA \rightarrow sA$, where gA is the guard. For the rest of the paper, we assume that any action A can be written in the form:

$A = gA \rightarrow sA$	*Guarded command*	(10)

such that:

$gd\ A = gA$	(11)

and

$gd\ sA = true$	(12)

Thus, enabledness of an action can be determined by checking its guard portion. Furthermore, we note that since $gd\ A = true$, we can derive the following property of sA by applying the definition of enabledness (formula 9):

wp (sA, *false*) = *false*	*Property sA*	(13)

Thereby sA must not establish a false post-condition. The weakest precondition semantics for an action $A = gA \rightarrow sA$ is:

wp ($gA \rightarrow sA$, q) = $gA \Rightarrow$ wp (sA, q)	*wp for guarded command*	(14)

Having defined the guarded actions, we can define conditional choice and repetitive construct:

wp (**if** A **fi**, q)	= wp (A, q) ∧ gA	*Conditional choice*	(15)
wp (**do** A **od**, q)	= (∀n.wp (A^n, $gA \vee q$)) ∧ (∃n.¬gA^n)	*Repetitive construct*	(16)

where A^0 = *skip* and $A^{n+1} = A^n; A$. The repetitive construct defines that each action enables some action or establishes q and that there must exist some that does not enable any other, i.e. partial correctness and termination. Consequently, an action A within **do** … **od** may execute only when its guard gA holds.

3.2 Inter-action Dependencies

Expressing that an action depends on another action can be modelled using a special operator. We denote it the *dependency operator* \\. Letting A and B be actions, where $A\backslash\backslash B$ denotes that $A \hookrightarrow B$, we define \\ to be:

Def. 1, dependency operator: $A\backslash\backslash B \quad = gA \wedge gB \rightarrow A; B$

Whenever having the construct $A\backslash\backslash B$, we call the dependent action A the *native* action and B the *trailing action*. For the gB to evaluate true after having executed A, we need to assure that A preserves gB by not assigning the free variables of B so that it would disable gB. This characteristic is shown in Section 3.3 by calculating the guard for $A\backslash\backslash B$, as is the wp for $A\backslash\backslash B$. Typically, at the time of termination $A\backslash\backslash B$ would be disabled, i.e. modelled to be executed exactly once.

3.3 Characteristics of the \\-operator

The dependency operator introduces some restrictions. We examine these by defining how \\ relates to the provided semantics of section 3.1 by exposing its characteristics by calculating its weakest precondition.

Characteristic 1, wp for \\: wp($A\backslash\backslash B$, q)

= wp ($gA \wedge gB \rightarrow A; B, q$)	// Def. 1
= $gA \wedge gB \Rightarrow$ wp ($A; B, q$)	// formula 14
= $gA \wedge gB \Rightarrow$ wp (A, wp (B, q))	// formula 5
= wp ($gA \wedge gB \rightarrow A$, wp ($B, q$))	// rewrite 14

This characteristic coins the meaning of the \\ operator. Initially gA and gB need to hold and after executing A, a state where B is enabled is reached, and after executing B, q is established. Hence, A must not disable B.

Assuming that B establishes q whenever gB holds and executed after A, we can calculate the collective guard of $A\backslash\backslash B$. This collective guard is deduced with the help of wp formulae.

Characteristic 2, guard of \\: g($A\backslash\backslash B$)

= ¬wp ($A\backslash\backslash B$, *false*)	// formula 8
= ¬wp ($gA \wedge gB \rightarrow A$, wp ($B$, *false*)	// charac. 1
= ¬($gA \wedge gB \Rightarrow$ wp (A, wp (B, *false*))	// formula 14
= ¬($gA \wedge gB \Rightarrow$ wp (A, wp ($gB \rightarrow sB$, *false*)))	// formula 10
= ¬($gA \wedge gB \Rightarrow$ wp (A, $gB \Rightarrow$ wp (sB, *false*))	// formula 14
= ¬($gA \wedge gB \Rightarrow$ wp (A, $gB \Rightarrow$ *false*))	// formula 13
= ¬($gA \wedge gB \Rightarrow$ wp (A, ¬$gB \vee$ *false*))	// Def. ⇒
= ¬($gA \wedge gB \Rightarrow$ wp (A, ¬gB))	// tautology
= ¬(¬($gA \wedge gB$) ∨ wp (A, ¬gB))	// Def. ⇒
= ¬¬($gA \wedge gB$) ∧ ¬wp (A, ¬gB)	// Def. deM
= $gA \wedge gB \wedge$ ¬wp (A, ¬gB)	// double neg

Characteristic 2 defines the general structure of the guard that must hold for the action $A\backslash\backslash B$ to be enabled. In short, the gA and gB need to hold and A must not disable B.

$A\backslash\backslash B$ is not commutative. This is the case partly because the significance of order i.e. distinction between the native action and trailing action in definition 1, where the native

action must not disable the trailing one. These characteristics support the definition provided. Hence, we conclude property 1 for non-interference:

> **Property 1, non-interference native:** The native action can only assign the free variables of the trailing action in a manner that does not disable the guard of the latter.

Property 1 states that a native action A must assure not to contribute towards disabling the trailing action B. Therefore, A is not allowed to arbitrarily assign the variables of B. However, the trailing action B can assign the native action's variables; otherwise, the impact of the trailing action would be restricted to the inclusion of the guard of the dependency relation.

4. Dependency on an action system level

As the fundamentals of an action and the characteristics of the dependency operator are provided, we extend usage of the operator to be used within an action system. The action system building blocks are defined in Section 4.1. We consider reactive action systems, where independent systems operate as a part of a more complex system. To navigate the complex system and depend on these remote action systems, we introduce a means for remote referencing in Section 4.2.

4.1. Action system at a glimpse

To start reasoning with action systems, we specify the elements of one, here named $\mathcal{A}$:

> **Def. 4, action system**: $\mathcal{A}$ = |[**var** *v*,*w** **proc** *P*:*p*; *R**:*r* ● *Init*: A_0; **do** *Op*: *A* **od**]| : *i*

In $\mathcal{A}$, *v* and *w** are the variables declared by this action system. Variables *v* are *local* and *w** constitute the *export*ed variables (denoted with an asterisk). Procedures are declared in the clause *proc* where *P*: *p* is a local procedure *p* named *P*, only executed if called upon whilst *R** is a globally referable procedure. Action *Init*:A_0 is the initialising action assigning the declared variables their initial value where *Init* is the label of this action. Each action label ∈ *Name of action labels* in the declaring action system. The **do** … **od** bracket pair constitute the repetitive construct (formula 16) within which the action *A* labelled *Op* is repeatedly executed until *A* aborts or until termination. Variables *i* stand for the optional *import*ed variables that are declared and exported by other action systems but referenced from this. Together, import *i* and export *w** constitute a situation resembling shared writable memory where the variable type is declared by the exporting action system.

Because considering reactive action systems the action system $\mathcal{A}$ is a part of a more complex system, where all other action systems are considered as $\mathcal{A}$'s environment, commonly denoted as $\mathcal{E}$. As the action atomicity holds on the whole complex system, any atomic action of $\mathcal{A}$ can be preceded by an action in $\mathcal{E}$ impacting $\mathcal{A}$ by writing to $\mathcal{A}$'s global variable space. Hence, the reactive component does not terminate by itself as the environment can, through the global variables, enable some actions within this. This makes the termination a global property and the formalism comes to showing properties of execution traces.

Any set of action systems in the reactive system can be composed to form a coherent monolithic action system. This is realised with the commutative and associative parallel composition operator ||, defined in Definition 2:

Def. 2, parallel composition '||': Let

$\mathcal{A}$ = |[**var** v_a, w_a*; **proc** *P*:*p* ● *Init*:A_0; **do** *Op*:*A* **od**]| : *i* and

$\mathcal{B}$ = |[**var** v_b, w_b*; **proc** *R**:*r* ● *Init*:B_0; **do** *Op*:*B* **od**]| : *j* then

$\mathcal{C} = \mathcal{A} \parallel \mathcal{B}$ = |[**var** x_m, x_n*; **proc** *P*:*p*; *R**:*r* ● *Init*:A_0; B_0;

do *Op*$\mathcal{A}$: *A* [] *Op*$\mathcal{B}$: *B* **od**]| : *h* where

$h = i \cup j \setminus (w_a \cup w_b)$, $x_n* = w_a \cup w_b$ and $x_m = v_a \cup v_b$ provided that $v_a \cap v_b = \emptyset$.

Definition 2 states that if a set of action systems operates on a disjoint set of local variables, $v_a \cap v_b = \emptyset$, procedure names and action labels, they can be composed without renaming to one action system where the actions within the repetitive **do** … **od** loop are treated non-deterministically and procedures remain intact. If the local variables are not disjoint or the local procedure names coincide; non-overlapping can be achieved through renaming whereas the action labels are be given a suffix indicating their origin. In the declaration above, action system $\mathcal{C}$ is a parallel composition of $\mathcal{A}$ and $\mathcal{B}$ where the possible execution traces remain unchanged. || has the immediate drawback of compromising modularity and reusability, i.e. composing action systems $\mathcal{A}||\mathcal{B} = \mathcal{C}$ does not guarantee that once decomposing $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{B}$ are recovered in their original form. Consequently, composition provides a means to form an abstract view of the system as well as refactoring the system.

4.2. Inter-Action System dependencies

Dependency within one action system is denoted by applying the dependency operator within the **do** … **od** construct with a reference to an action within this same repetitive construct. For referring to actions in the environment of this action system, means to make "remote dependency references" need to be defined. The trailing action operates in its own right, i.e. possibly providing its service to many disjoint actions without these having to be aware of each other. To reference a remote system providing this service, we define the @ reference:

Def. 3, @ reference: Let *B* be an action and $\mathcal{B}ar$ an action system where *B* ∈ actions of $\mathcal{B}ar$, then *B*@$\mathcal{B}ar$ refers to action *B* in action system $\mathcal{B}ar$.

The @ is a postfix to an action where *A**B*@$\mathcal{B}ar$ denotes action *A* to depend on action *B* in action system $\mathcal{B}ar$. Because of the atomicity of \\, action *A**B*@$\mathcal{B}ar$ waits for *B* to finish; calling this a *direct dependency* relation. Consequently, several actions can depend on *B*@$\mathcal{B}ar$ without interference as *B*@$\mathcal{B}ar$ is atomic and provides only to one system at any given time resembling the situation where *A* requests a resource possessed by *B*@$\mathcal{B}ar$.

Direct dependency relations do however halt the execution of the native system until the referenced action *B*@$\mathcal{B}ar$ terminates. As the scheduling for the whole system is not of interest, breaking the atomicity might be of interest. This can be done whenever *B*@$\mathcal{B}ar$ only enables another action, labelled B_{wake}@$\mathcal{B}ar$ that ought to be executed in the wake of *A*. Hence, if *B*@$\mathcal{B}ar$ enables B_{wake} and the system can guarantee that B_{wake} is executed at some point after the reference, the atomicity of \\ is broken. This lets the native system continue its execution until the possible results of B_{wake} are required. Thus,

$A\backslash\backslash B@\mathcal{B}ar$ only enables B_{wake} that is assured to execute prior to the first execution of the action labelled B_{nat}.

$\mathcal{A}$ = |[**var** v,w* **proc**; ● *Init*: A_0; **do** *Op*: $gOp \rightarrow A\backslash\backslash B@\mathcal{B}ar$
[] "*other actions*" **od**]| : i

$\mathcal{B}ar$ = |[**var** j,i* **proc** B*: $gB_{\text{orig}} \wedge k = false \rightarrow k := true$; ● *Init*: Bar_0;
do B_{wake}: $k = true \wedge gB_{\text{orig}} \rightarrow sB_{\text{orig}}$; C; $k := false$
[] B_{nat}: $k = false \wedge gB_{\text{orig}} \rightarrow sB_{\text{orig}}$
[] "*other actions*" **od**]| : l

Here actions labelled B_{nat} and B_{wake} assure sB_{orig} to be executed once but in addition to sB_{orig}, the referenced action labelled B_{wake} executes an additional action (possibly stuttering) C and disables itself through assigning k false. As the guard of the globally referable procedure B* is the same as for the alternatives of B_{nat} and B_{wake}, the semantics of the remote dependency is not altered. The impact on the system is similar to the one when considering only intra-action system dependencies but the atomicity is deliberately broken down with the Boolean of k for the sake of immediate progress, i.e. being able to execute the "*other actions*" in $\mathcal{A}$ before B_{wake} is finished. We call this *indirect dependency*. Moreover, if action system $\mathcal{A}$ is the only system importing variable k, then we say that k is a dedicated variable for this dependency. The trade-off with breaking the dependency is that the execution order cannot be guaranteed and it should hence be used carefully, i.e. as above when B* enables B_{wake}, $A\backslash\backslash(B\backslash\backslash C)$ ordering is not necessarily kept as A; B_{wake}; C because B_{wake} might actually execute after C, which is obvious as the atomicity was deliberately broken. However, it can easily be shown that A executes before C and before B_{wake}.

5. A short example

To clarify the realistic implementation scope of the ideas presented in this extended abstract, we outline a short, easily conceivable example. This example bases on fraction of a Buyer-Seller relation where the seller runs an ERP-system (Enterprise Resource Planning).

Assuming two actions $A \hookrightarrow B$, a realistic scenario could be that A wants to buy something sold by B i.e. a normal buyer-seller relation. Letting gA be 'has money' and A constitute the state update, gB could realistically be 'in stock' where B merely updates the stock. Consequently, per Definition 1, $gA \wedge gB \rightarrow A; B$, buyer A has money whilst the product is in stock and once bought the stock is updated. The novelty of this approach is that in order for A to execute, gB needs to be true, i.e. product must be in stock for the buyer to buy[1] and A need only to know the "interface" of B, i.e. sell if in stock.

$\mathcal{B}uyer$ = |[**var** v,w* **proc**; ● *Init*: A_0; **do** *Buy*: $gA \rightarrow A\backslash\backslash B@\mathcal{S}eller$ **od**]| : i
$\mathcal{S}eller$ = |[**var** j,i* **proc**; ● *Init*: B_0; **do** B: $gB \rightarrow B$ **od**]| : l

However, as there is no reason why the $\mathcal{B}uyer$ should halt until the $\mathcal{S}eller$ is complete, we break the atomicity.

$\mathcal{B}uyer$ = |[**var** v,w* **proc**; ● *Init*: A_0; **do** *Buy*: $gA \rightarrow A\backslash\backslash B@\mathcal{S}eller$ **od**]| : i

[1] Mathematically, knowing this a priori is irrelevant as the system can be modelled so that once the purchase is done, the product must be in stock.

$\mathcal{S}eller$ = |[**var** j,i* **proc** B*: $gB \wedge j = false \rightarrow j := true$; ● $Init$: B_0;
do B_{wake}: $gB \rightarrow B$; D
[]B_{nat}: $gB \rightarrow B$ **od**]| : l

Here the $\mathcal{S}eller$'s gB stand for "in stock". Hence, the dependency relation in the action labelled *Buy* is disabled unless gB evaluates true. The auxiliary action D executes only in the wake of a dependency reference to proc B* and could stand for shipping the product and invoicing. Naturally, this relation can be extended to a multi-phased dependency relationship where payment, cancellation, trust and other policies can be included.

6. Conclusions

As the future is likely going to be about navigating the ubiquity of information, being able to select, rely on and process relevant information, as well as to reason rigorously with these, the need to formally treat remote providers of this information is evident. In this extended abstract, we have shown research results in how inter-service dependencies can be modelled in the action system framework. We have introduced and briefly outlined the characteristics of a new dependency operator \\ and exemplified its feasibility in a short example.

As the research on the \\ operator is far from finished, future work will address chains of dependencies, non-ordered dependencies as well as refinement. The goal is to gain and present a library of well defined operators that address the challenges brought along with the ever increasing distribution of computations and responsibilities. We believe the service oriented architecture provides a feasible and demanding platform to verify results upon.

References

1. Neovius, M. and Yan, L.: A Design Framework for Wireless Sensor Networks. In Proceedings of World Computer Congress - WCC 2006, Ad Hoc networking track, 2006.
2. Roman, G.C., Julien, C., and Payton, J.: A formal treatment of context-awareness, In Proceedings of FASE'04, 2004.
3. Neovius, M. and Sere, K.: Formal Modular Modelling of Context-Awareness. To appear in Post-proceedings of FMCO 2008. LNCS, 2009.
4. Degerlund, F. and Sere, K.: A Framework for Incorporating Trust into Formal Systems Development. In Theoretical Aspects of Computing – ICTAC 2007, 4th International Colloquium, Proceedings, 2007.
5. Dijkstra, E. W.: A Discipline of Programming. Prentice Hall, 197632.
6. Back, R.J.R and Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In Proceedings of the 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 1983
7. Sere, K.: Stepwise derivation of parallel algorithms, PhD dissertation, Åbo Akademi 1990.
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM, vol. 18, no. 8, 1975.

A Contract-Based Approach to Adaptivity in User-Centric Pervasive Applications*

Martin Wirsing, Moritz Hammer, Andreas Schroeder, and Sebastian Bauer

Ludwig-Maximilians-Universität München, Germany
{wirsing, hammer, schroeda, bauerse}@pst.ifi.lmu.de

Abstract. Pervasive user-centric applications operate in highly dynamic and uncertain environments. Designing such applications as one monolithic component taking all possible environments into account inevitably leads to bad system design. We instead propose constructing partial solutions handling only a subset of all possible environments, and changing the system as the environment evolves. We use an assume-guarantee contract framework to infer the conditions under which configurations exhibit the desired functionality. Furthermore, we show how a system undergoing reconfigurations can be shown to satisfy a global assume-guarantee contract.

1 Introduction

Pervasive user-centric applications are applications running on systems seamlessly integrating with their environment and adapting it to the user's current emotional, cognitive, and physical state [6]. Such applications must deal with several challenges:

- They must operate in highly dynamic and uncertain environments.
- They often interact with other agents and humans.
- They must remain non-disruptive, and at the same time improve the experience of the user.

In this paper, we take a closer look on how the first challenge in the development of user-centric applications can be handled: dealing with highly dynamic and uncertain environments. Our approach involves making explicit assumptions about the environment and the functionality of the system using assume-guarantee contracts[3]. We annotate the components constituting the system with assume-guarantee contracts in order to verify whether a global contract can be satisfied. If this is not the case, we introduce components monitoring the system. By reconfiguring the system as soon as one of the monitored conditions get falsified, we achieve adaptivity within the bounds of a given specification defining the desired system behaviour. Furthermore, knowing the assume-guarantee pairs of components, the reconfiguration rules and the overall contract to satisfy, we can verify whether the system under construction satisfies the global system contracts that describe its overall purpose.

* This work has been partially supported by the EC project REFLECT, IST-2007-215893 and the GLOWA-Danube project 01LW0602A2 sponsored by the German Federal Ministry of Education and Research.

The remainder of this paper is structured as follows. In Section 2, we give an informal overview of our contract-based approach to adaptivity. Following this informal description, Section 3 gives an example scenario for the usage of our approach, before we introduce the formal assume-guarantee framework in Section 4. We conclude in Section 5.

2 Contract-Based Approach to Adaptivity

User-centric pervasive adaptive systems need to be able to cope with a large amount of uncertainty. Sensor readings might not be available or be inaccurate, and actuators might fail to achieve the desired effect on the user. Different users might respond differently to stimuli supplied by the system, and users might become bored by a stimulus after different numbers of repetitions. In order to cope with the challenges of such a volatile situation, such systems need to be engineered in a manner that supports adapting to changes of the user and the environment. Obviously, such adaptivity can be realised by implementing lots of case distinctions that address individual situations. However, as the number of problems that need to be addressed increases, this will lead to bloated and unmaintainable code.

Instead, we propose the use of *components* and *reconfiguration* to achieve adaptivity. Components are considered to be black boxes, making explicit only their communication requirements by means of *required* and *provided* ports. A system is comprised from a number of components and their *configuration*, which describes how the required ports are connected to suitable provided ports. Numerous component models describe how exactly this is achieved, and how the components can communicate in order to achieve a common goal [4]. For the REFLECT project, we have developed our own component framework [6], which is specially tailored to building adaptive systems with multimedia sensors and actuators.

In a component application, individual components provide parts of the functionality required by the entire system. By substituting, adding and removing individual components, the system's behaviour can be changed. This process is called *reconfiguration*. Since entire components are replaced, little code needs to be added to the components to achieve this kind of adaptivity. Instead, it is attained on a level more coarse: the level of the system architecture.

Still, reconfiguration is a difficult problem, and while many framework support it [5] the problems often outweigh the utility. One of this problems is the necessity to plan how reconfiguration should be conducted, which usually requires both anticipation of possible future problems as well as an algorithm to figure out a reconfiguration plan that operates on a component configuration that has possibly undergone many reconfigurations already. The anticipation of future reconfiguration scenarios is a difficult task. Here, we propose a way to support the planning of reconfiguration by annotating components with assume-guarantee pairs. Informally, the guarantee describes how a component will conduct communication over its ports, given that the assumption about the communication received by its ports holds. In this paper, we introduce a semantics-based assume-guarantee framework similar to the framework proposed in [2].

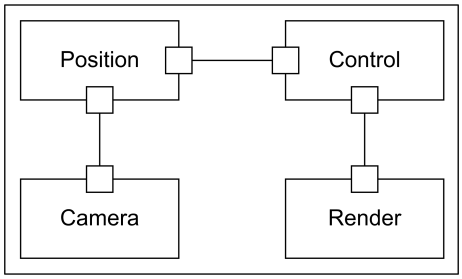


Fig. 1. Initial System

From such assume-guarantee specifications, a component's anticipation of possible employment scenarios can be derived. When designing a system, by connecting a component to other components, the composition can be checked against the guarantees provided by the communication partners, and possible mismatches can be detected. Checking composability in such a way has a long tradition (e.g., [1]). Furthermore, the intended behaviour of the entire application can be specified by assume-guarantee pairs, where the assumptions address the physical environment, and the guarantees the output produced. Again, the compatibility of a component configuration to such global specifications can be checked, and invalid application designs can be detected.

However, an invalid application configuration can still be useful under certain conditions. Identifying these assumptions and monitoring their validity allow to restrict the execution of the generally invalid application to valid situations at runtime. If the assumptions are about to get invalidated in a system run, it is still possible to execute a reconfiguration to a system that will show the desired behaviour in the new environment. The role of *monitors* is hence to allow to deploy system configurations needing assumptions that are not satisfied in the general case, and trigger reconfigurations as these assumptions get falsified.

3 Example Scenario: Adaptive Advertising

In order to illustrate our approach, we use a simple adaptive advertising scenario. The general idea of adaptive advertising is to adapt the displayed ad to the current situation in front of it – whether there are several people just passing by, a small group of persons watching the ad carefully, or just one person in front of it waiting for someone else. The system uses cameras to observe the passers-by, and by this enables the ad to react to e.g. the number of passers-by watching the advertisement, to discover their interest in the advertisement by analysing their gaze direction and exposure time, or to enable gesture-based interactions with a passer-by becoming interested in the ad.

A simple scenario within the vast ranges of possibilities the adaptive advertising setting offers is an adaptive car advertisement, in which the displayed car reacts to the position of users in front of the display: By moving around the display, a selected user controls the orientation of the car.

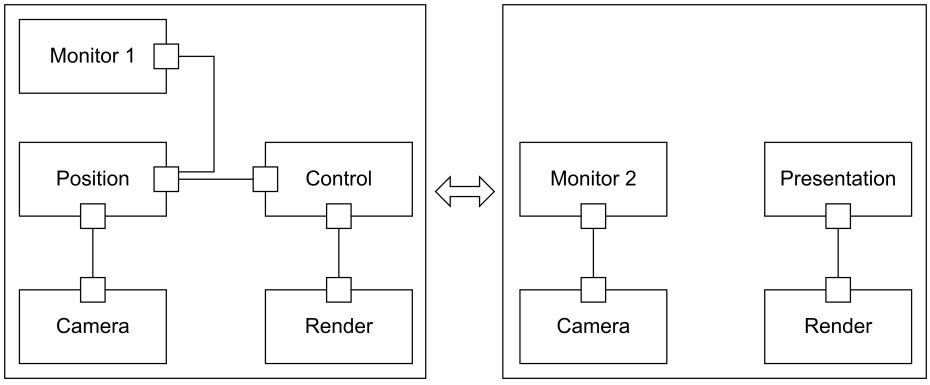


Fig. 2. System with Reconfiguration

The contract to be satisfied by this system consists of two guarantees: (G1) Being an interactive ad, the system should react to a user in front of the display. (G2) The content displayed must change at least every ten seconds: an advertising campaign using a large-scale display should not waste its capabilities by showing static content.

A first realisation of the system consists of four components (cf. Fig. 1): a camera component for image acquisition, a position detection component detecting the position of persons in front of the display, a control component selecting the person to be given control over the car movements and how his position should be related to the car's rotation, and finally a rendering component displaying content. This simple realisation is problematic, however. It cannot provide the guarantee that the displayed content changes every ten seconds, as the car movement depends on the control of a person in front of the display. Using a simple assume-guarantee calculus, we can show that the system is incomplete although it provides part of the desired functionality.

By introducing a monitor observing whether someone is in front of the display, the system can be made aware that it is about the violate its contract. Then, a reconfiguration can be triggered which alters the system such that it shows a constantly revolving car (cf. Fig. 2). In more formal terms, introducing a monitor allows to assume that the environment exhibits certain features (e.g. always have someone in front of the display) that it does not exhibit in the general case. Note that the second system (Fig. 2, right) also needs monitoring, as it again does not satisfy G1: The second system does provide interactive content to its viewers, and therefore must be changed as soon as a person is in front of the display.

In the following section, we introduce a formal framework allowing to check the properties described informally above, and show how it can be proven that the overall system satisfies the global contracts G1 and G2.

4 Assume-Guarantee Specifications

We annotate every component in our adaptive system by a pair of assertions (A, G). The assertion A formulates a property the component assumes from its environment

whereas G is the guarantee it provides to the environment given that A is satisfied. In this way every component can be proven to be correct with respect to their assume-guarantee specification.

We introduce a simple assume-guarantee framework which is formulated on the semantic domain in terms of runs over a given signature. A *signature* Σ consists of a set of provided and required ports, denoted by $ports_{prv}(\Sigma)$ and $ports_{req}(\Sigma)$ respectively. We assume the notion of a subsignature $\Sigma \subseteq \Sigma'$ and the supremum of two signatures $\Sigma \sup \Sigma'$; both notions are defined in the obvious way. A Σ-run is an abstract structure representing one possible behaviour of the system. One possibility – among others – to refine the notion of runs is to see them as capturing reception and sending of messages on ports (given by Σ) over (discrete or continuous) time.

Assumptions, guarantees as well as implementations of components are considered to be assertions. Given a signature Σ, a *Σ-assertion* E (also denoted by $E : \Sigma$) is identified with a set of Σ-runs. We assume that every Σ-assertion E can be lifted to a Σ'-assertion $E \uparrow^{\Sigma'}$ for $\Sigma \subseteq \Sigma'$. Moreover, we define the composition of assertions by $E_1 : \Sigma_1 + E_2 : \Sigma_2 := E_1 \uparrow^{\Sigma} \cap E_2 \uparrow^{\Sigma}$ for $\Sigma = \Sigma_1 \sup \Sigma_2$. From now on, signatures and liftings are omitted where they are not essential.

Assume-guarantee pairs are formulated as pairs of assertions $(A : \Sigma_A, G : \Sigma_G)$. Satisfaction is defined simply by inclusion of runs – more precisely, every run in M which is in A (i.e. satisfies A) must be in G.

Definition 1. *Let $M : \Sigma$ be an implementation, and $A : \Sigma_A$, $G : \Sigma_G$ two assertions. M satisfies $(A : \Sigma_A, G : \Sigma_G)$, denoted by $M \models (A, G)$, if and only if $\Sigma_A, \Sigma_G \subseteq \Sigma$ and $M \cap A \subseteq G$.*

Parallel composition of implementations preserves this satisfaction relation.

Lemma 1. *Let $M_1 : \Sigma_1$, $M_2 : \Sigma_2$ be two implementations, and let $ports_{prv}(\Sigma_1) \cap ports_{prv}(\Sigma_2) = \emptyset$ and $ports_{req}(\Sigma_1) \cap ports_{req}(\Sigma_2) = \emptyset$. If $M_1 \models (A_1, G_1)$ and $M_2 \models (A_2, G_2)$ and A is an assertion for which it holds $A \cap G_1 \subseteq A_2$, $A \cap G_2 \subseteq A_1$, and $A \subseteq A_1 \cup A_2$ holds, then $M_1 + M_2 \models (A, G_1 \cap G_2)$.*

When building component systems we want to check whether the resulting specification satisfies a global system specification. Therefore we introduce a refinement relation which allows assumptions to be weakened and guarantees to be strengthened.

Definition 2. *(A, G) refines (A', G'), denoted by $(A, G) \preceq (A', G')$, if $A' \subseteq A$ and $G \subseteq G'$.*

A major requirement for refinement relations is its compatibility with the satisfaction relation for implementations, i.e. whenever an implementation satisfies a refined contract, it satisfies the original contract.

Lemma 2. *If $M \models (A, G)$ and $(A, G) \preceq (A', G')$ then $M \models (A', G')$.*

A global system specification $(A_{\text{sys}}, G_{\text{sys}})$ can then be verified in the following way. Assume that the composed system $M_1 + \ldots + M_n$ satisfies (A, G), then in order to show that it also satisfies the global system specification $(A_{\text{sys}}, G_{\text{sys}})$ it suffices to show $(A, G) \preceq (A_{\text{sys}}, G_{\text{sys}})$.

In order to verify dynamic, reconfigurable systems, we must define the notions of a *composable* system that is *configured* at each moment by *connectors*. From now on, we consider more concrete runs of the form $\rho : \mathbb{R}_0^+ \to S$ with S a domain for states of runs.

Definition 3. *A set of implementations $M_1, \ldots, M_n$ is called* composable *iff $\Sigma_i \cap \Sigma_j$ is the empty signature for all $i \neq j$. A* configuration *of $M_1, \ldots, M_n$ is a set of runs C over $\Sigma \cup Con(\Sigma)$, $\rho : \mathbb{R}_0^+ \to \mathcal{P}(\Sigma \cup Con(\Sigma))$, where $\Sigma = \sup_{i=1}^n \Sigma_i$ is the supremum over all signatures, and $Con(\Sigma) = ports_{req}(\Sigma) \times ports_{prov}(\Sigma)$ is the set of all connectors. A configuration is* valid *iff for all $\rho \in C$. $(r, p) \in \rho(i)$ implies that $r, p \in \rho(i)$ or $r, p \notin \rho(i)$.* [1] *The composition under C, $(M_1 + \ldots + M_n)|_C$ is the set $C \cap (M_1 + \ldots + M_n) \uparrow^{\Sigma \cup Con(\Sigma)}$.*

Lifting $M_1 + \ldots + M_n$ to $\Sigma \cup Con(\Sigma)$ means the set of all runs in which each state of a run $\rho \in M_1 + \ldots + M_n$ was extended with each subset of connectors $Con(\Sigma)$.

A composable system is therefore a set of implementations that do not share any ports a priori, hence allowing a configuration to define the port connections through its runs (note that we consider only *valid* configurations in the following). Then, a composition under configuration is the set of runs that are compatible with the connections defined by a run in the configuration.

Example 1. We give a very small example for a composable system and a configuration in the following. Let $ports_{req}(\Sigma_1) = \{a\}, ports_{prv}(\Sigma_1) = \emptyset$ and $ports_{prv}(\Sigma_2) = \{b\}, ports_{req}(\Sigma_2) = \emptyset$ be two signatures. Let $M_1 = \{\rho_1, \rho_1'\} : \Sigma_1$ and $M_2 = \{\rho_2\} : \Sigma_2$ be properties such that

$$\rho_1(i) = \begin{cases} \{a\} \text{ if } 0 \leq i \leq 2 \\ \emptyset \quad \text{otherwise.} \end{cases} \qquad \rho_2(i) = \begin{cases} \{b\} \text{ if } 1 \leq i \leq 3 \\ \emptyset \quad \text{otherwise.} \end{cases}$$
$$\rho_1'(i) = \emptyset \text{ for all } i.$$

As the signatures of M_1 and M_2 are disjoint, M_1 and M_2 are composable. Let hence C be a configuration of M_1 and M_2 containing all ρ_c for which it holds that $(a, b) \in \rho_c(i)$ for $2 \leq i \leq 3$. Then, the composition $M = \{\rho\} = (M_1 + M_2)|_C$ consists of the single run ρ such that

$$\rho(i) = \begin{cases} \{a\} & \text{if } 0 \leq i < 1 \\ \{a, b, (a, b)\} & \text{if } 1 \leq i \leq 2 \\ \{b\} & \text{if } 2 < i \leq 3 \\ \emptyset & \text{otherwise.} \end{cases}$$

ρ is the composition of the runs ρ_1 with ρ_2 under one $\rho_c \in C$. Note that ρ_1' is not part of the composed system, as there is no run in M_2 to which it is compatible.

Note that Lemma 1 is still valid for composition under configuration, as the set of implementations is further constrained. It is also possible to take the configuration runs into the guarantee, as can be seen easily.

[1] In a more general setting we would require that the r and p are equal in value. Here however, we only consider predicates.

We now apply the described approach to our example scenario of Sect. 3. First, we must refine the general assume-guarantee framework to a real-time LTL logic given as follows.

Definition 4. *The set of RT-LTL-formulae is inductively defined by the grammar*

$$A ::= true \mid p \mid p_1 \sim p_2 \mid \neg A \mid A \vee A \mid \Box_t A$$

with ports $p, p_1, p_2 \in \Sigma$. Let $\rho : \mathbb{R}_0^+ \to \mathcal{P}(\Sigma \cup Con(\Sigma))$ be a valid run over Σ and $Con(\Sigma)$. Then $\rho \models A$ iff for all $i \in \mathbb{R}_0^+$. $\rho, i \models A$. The satisfaction relation $\models$ between pairs ρ, i and RT-LTL-formulae A is defined as follows.

1. $\rho, i \models true$.
2. $\rho, i \models p$ *iff* $p \in \rho(i)$.
3. $\rho, i \models p_1 \sim p_2$ *iff* $(p_1, p_2) \in \rho(i)$.
4. $\rho, i \models \neg A$ *iff* $\rho, i \not\models A$.
5. $\rho, i \models A \vee B$ *iff* $\rho, i \models A$ *or* $\rho, i \models B$.
6. $\rho, i \models \Box_t A$ *iff* $\forall i \leq j \leq i + t.\ \rho, j \models A$.

In the following, we use $\Box A$ as abbreviation for $\Box_\infty A$. $\Diamond_t A$ and $A \Rightarrow B$ are defined as usual. Note that $\Box A$ is equivalent to A.

Example 2. The system contract that must be shown for the system is the tuple

$$(GI, (\Box\Diamond_{10} CI) \wedge (\Box_{0.5} PT \Rightarrow \Diamond_1 R)).$$

Here, GI is "the camera produces a good image", CI "the image shown on the display changed", PT is "there is a person in front of the ad", and R stays for "the system responded to the person in front of the ad". Following Lemma 2, we must show that the contract guaranteed by the composition, (A, G), is a refinement of the system contract (A_{sys}, G_{sys}). For the sake of brevity, we will focus on discussing the system guarantee $\Box\Diamond_{10} CI$. In the example, the guarantee G of the composition (cf. Fig 2) can be shown to guarantee the following five properties, where C_1 is a shorthand notation for a conjunction of connection constraints defining "the system is in configuration 1" (cf. Fig. 2, left), and C_2 denotes similarly "the system is in configuration 2" (cf. Fig. 2, right).

1. $C_1 \Rightarrow PT \Rightarrow CI$
2. $C_2 \Rightarrow CI$
3. $C_1 \Rightarrow (\Box_3 \neg PT) \Rightarrow (\Diamond_{3.7} C_2)$
4. $C_2 \Rightarrow (\Box_{0.5} PT) \Rightarrow (\Diamond_{0.7} C_1)$
5. $C_1 \vee C_2$

Note that property 3 and 4 express changes in configurations: property 3 expresses that the first configuration (reacting to viewers) will be reconfigured in 3.7 seconds to the second configuration (showing animated content) if the scene in front of the display stays empty for three seconds. This property can be derived from the guarantee of Monitor 1 together with the connection properties of configuration one. Similarly,

property 4 expresses that the system will be changed within 0.7 seconds if a person is seen in front of the display for at least 0.5 seconds. Note that obviously, property 3 to 5 are derived from the behaviour of monitors and the specification of the configuration.

Properties 1 and 2 on the other side describe the behaviour of the system under each configuration: in configuration 1 (reacting to viewers), the displayed image changes if a person is in front of it. In configuration 2 (showing animated content), the image always changes.

With the help of a small realtime-LTL calculus, then, we can to show that these five properties imply $\Box\Diamond_{10}CI$, which is one step in showing the refinement $G \subseteq G_{sys}$. Altogether, it can be shown that the composition under the given reconfiguration refines the global contract. Hence, by switching between two initially insufficient configurations, the system is indeed showing the desired global behaviour.

5 Conclusion

Building pervasive user-centric applications brings several challenges, as they are operating in highly dynamic and uncertain environments. In this paper, we took a closer look in how this challenge can be taken by using a simple assume-guarantee-framework. The assume-guarantee approach allows to make assumptions of a configuration about the environment explicit. By monitoring these assumptions, it is possible to deploy an application that does not satisfy its contract in the general case, but only under given assumptions. As soon as these assumptions are violated, the system is reconfigured, so that the new configuration satisfies the contract under the now given assumptions.

An interesting future work is the introduction of probabilistic assume-guarantee contracts, as presented in [2]. Pervasive user-centric applications interface with the real world through sensors and actuators, which may be unreliable in and exhibit not only non-deterministic, but also probabilistic behaviour. With a probabilistic assume-guarantee framework, it would be possible to model this uncertain behaviour of the environment, and reason about the performance of pervasive user-centric applications in these environments.

References

1. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *8th European software engineering conference (ESEC '01)*, pages 109–120. ACM Press, 2001.
2. B. Delahaye and B. Caillaud. A Model for Probabilistic Reasoning on Assume/Guarantee Contracts. *ArXiv e-prints*, November 2008.
3. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
4. K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.
5. M. Sadjadi and P. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Computer Science and Engineering, Michigan State University, 2003.
6. Andreas Schroeder, Marjolein van der Zwaag, and Moritz Hammer. A Middleware Architecture for Human-Centred Pervasive Adaptive Applications. In *2nd Int. Conf. on Self-Adaptive and Self-Organizing Systems (PerAda '08)*, volume 0, pages 138–143, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

Passage retrieval and intellectual property in legal texts

Santiago Correa, Davide Buscaldi, Paolo Rosso
Natural Language Engineering Lab., EliRF Research Group
Dept. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, Spain
{scorrea, dbuscaldi, prosso}@dsic.upv.es, http://users.dsic.upv.es/grupos/nle

Alfonso Rios
Maat Knowledge, Spain
arios@mat-g.com, http://www.maat-g.com/

Question Answering (QA) can be viewed as a particular form of Information Retrieval (IR), in which the amount of information to return is the minimum required to satisfy the user needs expressed by a specific question such as: "Where is the Europol Drugs Unit?"[1]. A Passage Retrieval (PR) system is an IR system which, given a list of keywords (e.g.: "Electricity," "Motor", etc..) or a question such as the previous one, returns fragments of texts (passages) that are relevant to the user needs.

The *Cross-Language Evaluation Forum*[2] (CLEF), organises competitions for the assessment of multilingual information retrieval systems. In CLEF-2009 edition, due to the growing interest in natural language processing of legal texts from both the university and the business sector, tracks such as ResPubliQA[3] and IP[4] have been organised. We have participated in both tracks in the framework of the collaboration between the Natural Language Engineering Lab. of the Universidad Politécnica de Valencia (UPV) and the *Maat Knowledge* enterprise. In order to address both tracks on QA in legal texts and on Intellectual Property (IP) of patent retrieval, we have used the JIRS (JAVA Information Retrieval System) search engine, a freely available[5] PR system which has been developed at the UPV [1]. The results have been sent to the tracks organisers and will be presented at CLEF-2009 along with the ones of the other teams that have participated in the two tracks.

In the following sections we describe the main concepts of the JIRS system and how it has been applied to the ResPubliQA and IP tracks of CLEF-2009.

1. Passage retrieval system JIRS

Most of nowadays passage retrieval systems are not oriented to the specific question answering problem, because they only take into account the keywords of the question in order

[1] Question from ResPubliQA@CLEF-2009
[2] www.clef-campaign.org/
[3] http://celct.isti.cnr.it/ResPubliQA/
[4] http://www.ir-facility.org/the_irf/clef-ip09-track
[5] http://sourceforge.net/projects/jirs/

to obtain the relevant passages. JIRS is a PR system based on n-grams (an n-gram is a sequence of *n* adjacent words extracted from a sentence or a question.) JIRS is based on the premise that in a large collection of documents, an n-gram associated with a question must be found in this collection at least once. The PR system has the ability to find structures of questions in a large collection of documents quickly and efficiently through the use of different n-grams models. JIRS searches for all possible n-grams of the question in the collection and it quantifies them in relation to the n-grams quantity and weight that appear in these passages. For example, let us suppose that we have a database of publications of a newspaper. Using the JIRS system we aim at finding in the document of the collection an answer to a question such as "Who is the president of Colombia?". For instance, the system could retrieve the following two passages: "... Álvaro Uribe is the president of Colombia ..." and "...Giorgio Napolitano is the president of Italy...". Of course, the first passage should be given more importance because it contains the 5-gram "is the president of Colombia", whereas the second passage contains only the 4-gram "is the president of". In order to calculate the n-gram weight of each passage, first of all we need to identify the most relevant n-gram and assign to it a weight equal to the sum of the weights of all its terms. The weight of each term is set to:

$$w_k = 1 - \frac{\log(n_k)}{1+\log(N)} \quad (1)$$

Where n_k is the number of passages in which the term appears and N is the total number of passages. A more detailed description of the system JIRS can be found in [1].

2. Passage retrieval for question answering

ResPubliQA@CLEF-2009 competition address the problem of question answering in legal texts. Given a pool of 500 independent natural language questions, each system must return the passage (not the exact answer) which answers each question from the JRC-Acquis[6] collection of EU documentation where both questions and documents are translated and aligned for a subset of languages.

In order to use the JIRS system in this QA track, we had to analyse and transform the documents of the collection for indexing them in the JIRS search engine. The collection of the competition is made of documents in XML format, each one divided into paragraphs delimited by the tag <p>. Therefore, each paragraph has been defined as a document, tagged with the name of the document where it is contained and the paragraph number that corresponds to it. Once all the documents have been extracted from the collection, they have been indexed in JIRS according to the language that has been analysed. Once obtained the database indexed by JIRS, we had searched for the answer to each question of the track (see example in the previous page). For each question, the system has returned a list with the most likely documents where an answer to the question was found, according to the way JIRS works. In an additional experiment, we made use of the parallel collection provided for the competition by obtaining a list of answers in different languages (Spanish, English, Italian and French), choosing as best answer the one better ranked by JIRS and then translating all of them to a single language. A detailed description of how the system JIRS has been used in this QA track can be found in [2].

[6] http://langtech.jrc.it/JRC-Acquis.html

3. Passages retrieval for intellectual property

The CLEF IP track is coordinated by *Information Retrieval Facility*[7] (IRF) and *Matrixware*[8]. Its aim is to investigate IR techniques for patent retrieval in order to search for the prior art of a patent on a certain topic in order to determine whether or not a certain degree of plagiarism of ideas occurred. The track provided a collection of more than 1M patent documents, mainly derived from European Patent Office sources, in three languages: English French and German. Queries and relevance judgements have been produced manually by Intellectual Property experts (using a set of queries given by themselves) and automatically, using patent citations from seed patents.

The set of 500 patents in *xml* format contained information that was not useful. Therefore, the first step has been to eliminate this type of information. In addition, patents have a identification number which makes them unique, although it is possible to find different versions of a unique patent. Therefore, we had to eliminate this kind of repeated information. Last, we had to remove stop words from the documents. At the end of this pre-process, we obtained a smaller size collection, which could be indexed by the JIRS search engine. To ask JIRS for the related patents, we had to build the related words sequence to each patent, considering from each of the 500 patents its title and its relevant terms obtained after a summarization technique [3]. For each patent, the information of the title and its relevant terms was concatenated and given to JIRS as a words sequence.

A detailed description of how the system JIRS has been used in the task can be found in [4].

Acknowledgement

The work of the first author has been possible thanks to a scholarship funded by Maat Knowledge in the context of the project with the Universidad Politécnica de Valencia *Módulo de servicios semánticos de la plataforma G*. We also thanks the TEXT-ENTERPRISE 2.0 TIN2009-13391-C04-03 research project.

References

1. Buscaldi D., Rosso P., Gómez J.M., Sanchis E. Answering Questions with an n-gram based Passage Retrieval Engine. Journal of Intelligent Information Systems (82), 2009 (in press).
2. Correa S., Buscaldi D., Rosso P. NLEL-MAAT at CLEF-ResPubliQA. Proc. 10th Int. Cross-Language Evaluation Forum CLEF-2009 working notes (to be published in September 2009).
3. Hassan S., Mihalcea R., Banea C., Random-Walk TermWeighting for Improved Text Classification. IEEE International Conference on Semantic Computing, ICSC-2007, 2007.
4. Correa S., Buscaldi D., Rosso P. NLEL-MAAT at CLEF-IP at CLEF-ResPubliQA. Proc. 10th Int. Cross-Language Evaluation Forum, CLEF-2009 working notes (to be published in September 2009)

[7] http://www.ir-facility.org/the_irf/
[8] http://www.matrixware.com/