Grant Agreement N^{o} 215483

Title:	*Initial Set of Principles, Techniques and Methodologies for Assuring End-to-end Quality and Monitoring of SLAs*
Authors:	*UniDue, Tilburg, FBK, INRIA, Lero-UL, POLIMI, TUW, UPM, USTUTT*
Editor:	*Michael Parkin (Tilburg)*
Reviewers:	*Andreas Metzger (UniDue)*
Identifier:	*CD-JRA-1.3.4*
Type:	
Version:	*1.0*
Date:	*March 17, 2010*
Status:	*Final*
Class:	*Internal*

Management summary

This document is the compilation of the set of papers used to produce the ‘paper-based’ deliverable CD-JRA-1.3.4, which describes an initial set of principles, techniques and methodologies for assuring the end-to-end quality and monitoring of SLAs.

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement n 215483 (S-Cube).

Members of the S-Cube consortium:

University of Duisburg-Essen (Coordinator) – UniDue	Germany
Tilburg University – Tilburg	Netherlands
City University London – CITY	U.K.
Consiglio Nazionale delle Ricerche – CNR	Italy
Center for Scientific and Technological Research – FBK	Italy
French Natl Institute for Research in Computer Science and Control – INRIA	France
The Irish Software Engineering Research Centre – Lero	Ireland
Politecnico di Milano – Polimi	Italy
MTA SZTAKI – Computer and Automation Research Institute – SZTAKI	Hungary
Vienna University of Technology – TUW	Austria
Universit Claude Bernard Lyon – UCBL	France
University of Crete – UOC	Greece
Technical University of Madrid – UPM	Spain
University of Stuttgart – USTUTT	Germany
University of Amsterdam – VUA	Netherlands
University of Hamburg – UniHH	Germany

Published S-Cube documents

These documents are all available from the S-Cube Web Portal at `http://www.s-cube-network.eu/`

Contents

Embedding Continuous Lifelong Verification in Service Life Cycles[*]

Domenico Bianculli
University of Lugano
Faculty of Informatics
Lugano, Switzerland
domenico.bianculli@lu.unisi.ch

Carlo Ghezzi
Politecnico di Milano
DEEP-SE group - DEI
Milano, Italy
carlo.ghezzi@polimi.it

Cesare Pautasso
University of Lugano
Faculty of Informatics
Lugano, Switzerland
cesare.pautasso@unisi.ch

Abstract

Service-oriented systems are an instantiation of open world software, which is characterized by high dynamism and decentralization. These properties strongly impact on how service-oriented systems are engineered, built, and operated, as well as verified. To address the challenges of applying verification to open service-oriented systems, in this position paper we propose to apply verification across the entire life cycle of a service and introduce a verification-oriented service life cycle.

1. Introduction

One of the defining properties of Software Service Engineering, as opposed to traditional Software Engineering, is the *open world* assumption [3]. The implications of this assumption affect all aspects of this emerging discipline, e.g., from how to design a service-oriented architecture to the notion of correctness and quality that can be applied to define and check the integrity and the validity of a system design. In this position paper we focus on the consequences of this assumption on the way verification is carried out in the service life cycle.

In particular, we argue for the need for embedding continuous lifelong verification across the entire service life cycle. Thus, it is no longer sufficient to apply verification during a particular phase or using a specific technique, such as proving properties based on a service contract at design time, or testing a service at deployment time to ensure and guarantee that it will continue to work as it has been specified or tested during the remainder of its lifetime. Since services live in an open world, where change is frequent, unexpected, and welcome [13], it becomes important to be able to assert properties that have a *lifelong* validity: both at design time, deployment time, and at run time. *Continuous* verification is about checking services as they are put into production, but also advocates that they are monitored (and checked) during their entire productive life.

This paper shows how different verification techniques [2] can be applied at different stages of the service life cycle, by proposing to enhance conventional life cycle models with a verification-oriented life cycle layer. This verification-oriented life cycle iteratively integrates and correlates different techniques, and makes possible to guarantee that the same properties verified based on services contract specifications, will still be checked once the service becomes an executable artifact and it is embedded into a larger service-oriented architecture. This way, continuous lifelong verification can provide a fundamental building block for delivering self-adaptive systems [8].

The rest of this paper is organized as follows. Section 2 briefly surveys the existing proposals for service life cycles and discusses to which extent they support verification, leading to Section 3, which motivates the need for continuous lifelong verification of service-based applications. Section 4 presents our verification-oriented model of service life cycle, which shows how to achieve continuous lifelong verification. Section 5 discusses related work and Section 6 concludes the paper with an outlook on possible future research directions.

2. Life cycle models

In this section we discuss how service verification is embedded in the software service life cycle models proposed by several leading SOA vendors and SOC researchers.

Some authors (e.g., [10]) suggest to comply with a traditional life cycle, following a sequence of phases such as *analysis*, *design*, *development*, *testing*, *deployment* and *administration*. This model has a dedicated phase for (functional) testing, which strictly follows the development phase and precedes deployment. Hints are also given about the

[*]This work has been partially supported by the EU-FP7-215483 project "S-Cube" and by the ERC grant 227977 "SMScom".

possibility to monitor the service usage during the administration phase, even though the topic is not further discussed.

In other cases, the life cycle model is adapted to the specific characteristics of service-centric software systems, as proposed by [7]. This is the case for life cycle models proposed by industrial vendors. IBM, for example, proposes a *model*, *assembly*, *deploy* and *manage* cycle, in which the last phase is also devoted to monitoring the performance of a service and detecting the failure of system components. Sun presents the SOA Repeatable Quality methodology, which includes the *conception*, *inception*, *elaboration*, *construction* and *transition* phases in an iterative way. However, verification activities are not explicitly mentioned in the documentation. Oracle/BEA proposes a life cycle model that clearly separates design-time activities (i.e., *identify business process*, *service modeling*, *build and compose*) from those carried out at run time (i.e., *publish and provision*, *integrate and deploy*, *secure and manage*, *evaluate*). In this model, verification activities belong to the *secure and manage* phase, and are mainly focused on SLA management, performance optimization and dealing with error events.

As for academic contributions, [14] illustrates a (Web) service life cycle model and a service-oriented design methodology. The life cycle starts with an initial *planning* phase, followed by a set of phases to be iteratively repeated: *analysis and design*, *construction and testing*, *provisioning*, *deployment*, *execution and monitoring*. Verification is performed before services are put into operation, by means of functional, performance, interface and assembly testing, and when services becomes operational, by means of QoS monitoring techniques. [11] proposes a stakeholder-driven life cycle, with much emphasis on the assignment of activities to stakeholders and on the interaction between and across them. The service provider is responsible for service functional testing at design time and service monitoring at run time. From the point of view of service consumers, verification activities include application testing at design time, in case the service consumer plays also the role of an application provider/service integrator, and — at run time — monitoring of the services that are consumed.

3. Motivation

As shown above, existing proposals of service life cycles advocate to perform verification either at a specific stage of the life cycle, e.g., at design time, before putting a service into operation, or at execution time, while the service is being provisioned. In some cases, verification is performed at both stages, but usually the properties verified at one stage are different from and not related to the ones verified at the other stage, e.g., by following the classical dichotomy between functional and non-functional properties.

Such narrow scope of verification does not entirely address the implications of the open world assumption on how verification activities are performed. Indeed, the key issue of continuous lifelong verification consists of spanning verification activities across the service life cycle, which can be iterated multiple times. We motivate the need for applying continuous verification during the entire life span of a service by means of the following three statements.

Design-time verification only gives limited guarantees. This kind of verification is carried out by assuming some properties about and using some models of the environment with which the service will interact. The environment is represented mostly by 3rd-party services and the distributed network infrastructure. As for the former, there is no guarantee that a service provider will eventually fulfill the obligations promised in a service agreement. For example, during a standard maintenance activity, a provider could inadvertently modify an existing service into an upgraded but incorrect and/or incompatible version, which could break the compatibility with service clients. Additionally, a malicious provider could modify the exported service, by offering a lower-quality service than the one promised through the agreement. Regarding the latter, the parameters estimations used to model the network infrastructure are often inaccurate, since they must be provided *a-priori* by domain experts and are related to quantities that may change over time. Thus a service that before deployment was proved to satisfy the requested quality of service requirements may turn out in practice to violate them, because of the mismatch between the abstract models that were used for verification before the deployment and the actual state of the environment at run time.

A service lives also between design and execution. A service life cycle contains other stages besides design and execution. Thus a coherent lifelong verification methodology should indicate the use of specific techniques at every stage of the life cycle. For example, when a service is about to be deployed, it should be "auditioned" [5], i.e., it should be tested during the interaction with the actual services that will be provided by business partners, before exposing the service to public usage.

Execution-time verification can close the loop of iterative service life cycles. Some verification activities may operate on live data acquired by means of run-time monitoring, but it may not always be practical to perform them on-the-fly. Therefore such data can be used to fuel the next iterations of the service life cycle, by providing valuable feedback to the service architects, modelers, and developers. Live data can be exploited for a thorough audit, an accurate analysis and model calibration before deploying a new version of the service.

4. Verification-oriented life cycle

Since many of the life cycle models reviewed in Section 2 have a coarse-grained modularity in terms of activities [11], we ground our verification-oriented life cycle on a slightly modified version of the model described in [14]. This model has an intrinsic iterative structure of the life cycle, which is a prerequisite to achieve continuous lifelong verification. As we are going to show, it is thanks to the feedback provided by the verification activities that the loop in the iterative life cycle can be closed. With respect to the original formulation of the model, we shifted the *testing* and *monitoring* phases into the verification-oriented layer of the model.

The complete model is shown in Figure 1: non verification-related phases of the life cycle are depicted with a white-filled shape, while the corresponding verification activities are highlighted using a grey-filled shape.

The *analysis and design* phase identifies the requirements of the service-based application, builds the models of the business processes defining the applications and specifies the required services. We envision that at this stage, the requirements are captured and automatically formalized, such that their formal models will be made available across the entire lifespan of the service. This is a crucial step, since the formalized requirements represent the properties for which we want to assert their lifelong validity. A relevant contribution to this step is embodied by a modeling/specification language that should be able to capture all facets of the behavior and of the QoS related to services execution and interaction. The first step of continuous lifelong verification is to ascertain that the properties (both functional and non-functional) corresponding to requirements are met by abstract design models of the service, evaluated in a given context. To achieve this, we suggest the use of *static analysis* techniques such as model checking. The goal is to check if a formal description of a certain property holds in the service model by transforming it to a suitable representation that can be efficiently verified.

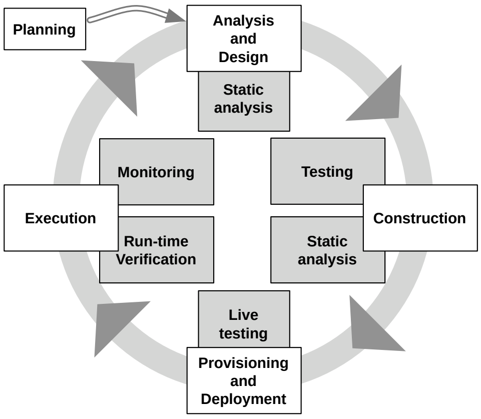


Figure 1. Verification-oriented life cycle

The *construction* phase is about developing the actual implementation of the service-based application. In this phase, we recommend to use *static analysis* techniques to verify the correctness of the application, as well as *testing* techniques that operate directly on the implementation. With respect to the previous phase, the main difference lies in the system under verification: in this phase, verification is performed against the concrete implementation of the service. Therefore, the verification techniques require more tuning; e.g., source-level analysis tools should be used. However, many of the verification artifacts used in the previous phase (e.g., temporal logic formulae or queueing network models) can still be used in this phase. To do so, the previously mentioned verification techniques can be used in combination: e.g., the counter-examples obtained from the execution of model checking can be used to derive dedicated test cases. Moreover, the properties captured during the analysis phase can be converted, using specific model-transformation techniques, into proper artifacts suitable for source-level verification and testing.

The *provisioning* phase is about making strategic decisions on service governance, certification, metering and billing, while the *deployment* phase is concerned about making the service publicly available. During these phases we claim that pre-deployment live testing, i.e., testing the interactions of a service with other actual services, can be a supporting tool for ensuring the validity of governance decisions, such as the choice of business partners to interact with, or the kind of service level agreement to be signed. Moreover, testing (both functional and non-functional) with the actual services may also reveal some problems that were not detected in the previous phase because of the abstract models that were used. Also in this phase, the original verification artifacts corresponding to requirements can be transformed into properties that can checked and/or measured during the live testing.

Finally, the *execution* phase is about the productive part of the cycle, when the service is kept in operation. In this phase, the main verification activities are monitoring and run-time verification. The former collects and analyzes data about the quality of the provisioned service and the 3rd-party services it interacts with, while the latter analyzes the execution trace to detect possible violations of the initial requirements. Indeed, the models corresponding to these requirements, are transformed, during the deployment, into data collectors (e.g., to monitor QoS properties such as response time and throughput) and failure detectors (e.g., to detect a violation of a functional or temporal assertion). One of the key aspects of continuous lifelong verification is that run-time verification activities may support model calibration, by using the collected data to provide a better estimation of the parameters that define the external operating environment [9], and *post-mortem* analysis, which is usually performed off-line by means of a static analysis tool. The information collected by these verification activities is also fed into the next iteration of the cycle providing valuable input to the *analysis and design* phase.

5. Related work

Section 2 has already described the different service life cycle models and how they deal with the issue of verification. However, there have been some other proposals about the verification of service-based applications, which do not strictly fall under the umbrella of service life cycles. For example, in previous work [6] some of the authors addressed the issue of lifelong verification of service compositions and proposed a methodology to deal with this issue. However, this proposal considered a simplified model of a very generic service life cycle, which only distinguished between a design-time phase and a run-time phase. Agile methods like Test-driven development [4] (TDD) emphasize the role of continuous testing during the development process. In principle, this is similar to our approach towards continuous verification; however they focus only on the specific implementation/development phase of the life cycle. ASOP [15] is a proposal for an agile service-oriented (development) process; however, as far as verification is concerned, it only considers TDD-like verification techniques.

The methodology followed by this work to create a service life cycle tailored to specific viewpoints has been introduced by [13], where a change-oriented service life cycle has been proposed in the context of service evolution featuring support for: (re-)configuration, alignment, and control of services upon changes. A similar approach — even though not specific to service-oriented systems engineering — can be found in [12], where a waterfall software development life cycle has been extended to include security into every phase of the life cycle.

6. Conclusion and future work

Service-oriented systems live in an open world that requires new engineering methodologies to deal with the intrinsic dynamic and decentralized nature of these systems. This paper has focused on verification, by showing how existing service life cycle models are inadequate to support continuous lifelong verification of service-based applications — which we believe to be a key aspect of this kind of applications — and it has proposed a verification-oriented life cycle to achieve this goal.

Although in this paper we have grounded our verification-oriented life cycle on a specific life cycle model, in the future we will consider other existing life cycle models and try to overlay the proposed verification-oriented life cycle on them. In particular, we will investigate the challenges to adapt the verification-oriented life cycle layer to agile development processes, since our proposal augments an iterative model. In all cases, we will evaluate its adoption in the context of real service-based applications development, by measuring the impact on the project quality, duration and cost. Moreover, we will investigate the role and the use of models [1] in the context of the proposed life cycle, to define a model-driven methodology to achieve continuous lifelong verification.

References

[1] D. Ardagna, C. Ghezzi, and R. Mirandola. Rethinking the use of models in software architecture. In *Proc. of QoSA 2008*, volume 5281 of *LNCS*, pages 1–27. Springer, 2008.

[2] L. Baresi and E. Di Nitto, editors. *Test and Analysis of Web Services*. Springer, 2007.

[3] L. Baresi, E. D. Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[4] K. Beck. *Test Driven Development by Example*. Addison-Wesley Professional, November 2002.

[5] A. Bertolino, L. Frantzen, and A. Polini. Audition of web services for testing conformance to open specified protocols. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 1–25. Springer, 2006.

[6] D. Bianculli and C. Ghezzi. Towards a methodology for lifelong validation of service compositions. In *Proc. of SDSOA 2008*, pages 7–12. ACM, 2008.

[7] M. B. Blake. Decomposing composition: Service-oriented software engineers. *IEEE Software*, 24(6):68–67, June 2007.

[8] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3):313–341, 2008.

[9] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburelli. Model evolution by run-time adaptation. In *Proc. of ICSE'09*. IEEE Computer Society, 2009. to appear.

[10] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, August 2005.

[11] Q. Gu and P. Lago. A stakeholder-driven service life cycle model for SOA. In *Proc. of IW-SOSWE'07*, pages 1–7, 2007.

[12] A. M. Hoole, I. Simplot-Ryl, and I. Traore. Integrating contract-based security monitors in the software development life cycle. In *Proc. of FLACOS'08*, pages 25–30, 2008.

[13] M. P. Papazoglou. The challenges of service evolution. In *Proc. of CAiSE 2008*, volume 5074 of *LNCS*, pages 1–15. Springer, 2008.

[14] M. P. Papazoglou and W. V. D. Heuvel. Service-oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, year 2006.

[15] A. Qumer and B. Henderson-Seller. ASOP: An agile service-oriented process. In *New Trends in Software Methodologies, Tools and Techniques*, pages 83–92. IOS Press, 2007.

A Guided Tour through SAVVY-WS: a Methodology for Specifying and Validating Web Service Compositions

Domenico Bianculli[1], Carlo Ghezzi[2], Paola Spoletini[3],
Luciano Baresi[2], and Sam Guinea[2]

[1] University of Lugano
Faculty of Informatics
via G. Buffi 13, CH-6900, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch
[2] Politecnico di Milano
DEEP-SE Group - Dipartimento di Elettronica e Informazione
piazza L. da Vinci, I-20133, Milano, Italy
{carlo.ghezzi, sam.guinea, luciano.baresi}@polimi.it
[3] Università dell'Insubria
Dipartimento di Scienze della Cultura, Politiche e dell'Informazione
via Carloni 78, I-22100, Como, Italy
paola.spoletini@uninsubria.it

Abstract. Service-Oriented Architectures are emerging as a promising solution to the problem of developing distributed and evolvable applications that live in an open world. We contend that developing these applications not only requires adopting a new architectural style, but more generally requires re-thinking the whole life-cycle of an application, from development time through deployment to run time. In particular, the traditional boundary between development time and run time is blurring. Validation, which traditionally pertains to development time, must now extend to run time. In this paper, we provide a tutorial introduction to SAVVY-WS, a methodology that aims at providing a novel integrated approach for design-time and run-time validation. SAVVY-WS has been developed in the context of Web service-based applications, composed via the BPEL workflow language.

1 Introduction

Software systems have been evolving from having static, closed, and centralized architectures to dynamically evolving distributed and decentralized architectures where components and their connections may change dynamically [1]. In these architectures, *services* represent software components that provide specific functionality, exposed for possible use by many clients. Clients can dynamically discover services and access them through network infrastructures. As opposed to the conventional components in a component-based system, services are developed, deployed, and run by independent parties. Furthermore, additional services can be offered by service aggregators composing third-party services to provide new added-value services.

This emerging scenario is *open*, because new services can appear and disappear, *dynamic*, because compositions may change dynamically, and *decentralized*, because no single authority coordinates all developments and their evolution.

Service-Oriented Architectures (SOAs) have been proposed to support application development for these new settings. An active research community is investigating the various aspects for service-oriented computing; research progress is documented, for example, by the International Conference on Service-Oriented Computing [2]. Several large research projects have also been funded in this area by the European Union, such as, amongst many others, SeCSE [3], PLASTIC [4], and the S-Cube [5] network of excellence, in which the authors are involved. The European Union has also promoted many initiatives to foster services-based software development and research, such as NESSI [6].

We strongly believe that a holistic approach is necessary to develop modern dynamic service-based applications. A coherent and well-grounded methodology must guide an application's life cycle: from development time to run time. SAVVY-WS (Service Analysis, Verification, and Validation methodologY for Web Services) is intended to be a first attempt to contribute to such a methodology, by focusing on lifelong verification of service compositions, which encompasses both design-time and run-time verification. SAVVY-WS is tailored to Web service technologies [7, 8]. The reason of this choice is that although SOAs are in principle technology-agnostic and can be realized with different technologies —such as OSGi, Jini and message-oriented middleware— Web services are the most used technology to implement SOAs, as corroborated by the many on-going standardization efforts devoted to support them. SAVVY-WS has been distilled by research performed in the context of several projects, most notably the EU IST SeCSE [3, 9] and PLASTIC [4] projects, and the Italian Ministry of Research projects ART DECO [10] and DISCoRSO [11]. A preliminary evaluation of the use of SAVVY-WS has been reported in [12–14]. SAVVY-WS is supported by several prototype tools that are currently being integrated in a comprehensive design and execution environment.

This paper provides a tutorial introduction to SAVVY-WS. Section 2 briefly summarizes the main features of the BPEL language, which is used for service compositions, and the ALBERT language, which is used to formally specify properties. Section 3 gives an overview of SAVVY-WS, which is based on ALBERT, a design-time verification environment based on model checking, and a run-time monitoring environment. Section 4 introduces two running examples that will be used throughout the rest of the paper. Section 5 discusses how ALBERT can be used as a specification language for BPEL processes. Section 6 shows how verification is performed at design time via model checking, while Sect. 7 shows how continuous verification of the service composition can be achieved at run time. Section 8 discusses the related work. Section 9 provides some final conclusions.

2 Background Material

2.1 BPEL

BPEL —Business Process Execution Language (for Web Services)— is a high-level XML-based language for the definition and execution of business processes [15]. It supports the definition of workflows that provide new services, by composing external Web services in an orchestrated manner. The definition of a workflow contains a set of global variables and the workflow logic is expressed as a composition of *activities*; variables and activities can be defined at different visibility levels within the process using the *scope* construct.

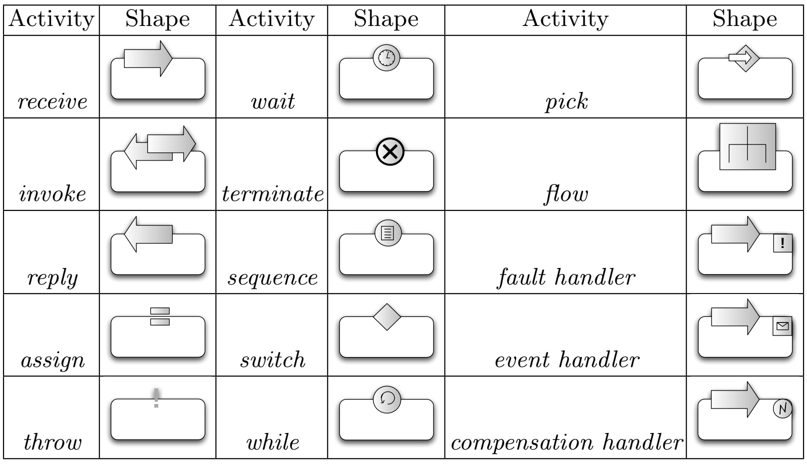
Activities include primitives for communicating with other services (*receive*, *invoke*, *reply*), for executing assignments (*assign*) to variables, for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of a process (*terminate*). Moreover, conventional constructs like *sequence, while,* and *switch* provide standard control structures to order activities and to define loops and branches. The *pick* construct makes the process wait for the arrival of one of several possible incoming messages or for the occurrence of a time-out, after which it executes the activities associated with the event.

The language also supports the concurrent execution of activities by means of the *flow* construct. Synchronization among the activities of a *flow* may be expressed using the *link* construct; a link can have a guard, which is called *transitionCondition.* Since an activity can be the target of more than one link, it may define a *joinCondition* for evaluating the *transitionCondition* of each incoming link. By default, if the *joinCondition* of an activity evaluates to false, a fault is generated. Alternatively, BPEL supports *Dead Path Elimination*, to propagate a false condition rather than a fault over a path, thus disabling the activities along that path.

Each *scope* (including the top-level one) may contain the definition of the following handlers:

- An *event handler* reacts to an event by executing —concurrently with the main activity of the *scope*— the activity specified in its body. In BPEL there are two types of events: message events, associated with incoming messages, and alarms based on a timer.
- A *fault handler* catches faults in the local *scope.* If a suitable *fault handler* is not defined, the fault is propagated to the enclosing *scope.*
- A *compensation handler* restores the effects of a previously completed transaction. The *compensation handler* for a *scope* is invoked by using the *compensate* activity, from a *fault handler* or *compensation handler* associated with the parent *scope.*

The graphical notation for BPEL activities used in the rest of the paper is shown in Fig. 1; it has been devised by the authors and it is freely inspired by BPMN [16].



Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

Fig. 1. Graphical notation for BPEL

2.2 ALBERT

ALBERT [12] is an assertion language for BPEL processes, designed to support both design-time and run-time validation.

ALBERT formulae predicate over *internal* and *external* variables. The former consist of data pertaining to the internal state of the BPEL process in execution. The latter are data that are considered necessary to the verification, but are not part of the process' business logic and must be obtained by querying external data sources (e.g., by invoking other Web services, or by accessing some global, persistent data representing historical information).
ALBERT is defined by the following syntax:

$$
\begin{aligned}
&\phi ::= \chi \;\mid\; \neg\phi \;\mid\; \phi \wedge \phi \;\mid\; (\;\mathsf{op}\ \mathsf{id}\ \mathtt{in}\ \mathsf{var}\ ;\ \phi\;) \;\mid\\
&\mathit{Becomes}(\chi) \;\mid\; \mathit{Until}(\phi,\phi) \;\mid\; \mathit{Between}(\phi,\phi,K) \;\mid\; \mathit{Within}(\phi,K)\\
&\chi ::= \psi\ \mathsf{relop}\ \psi \;\mid\; \neg\chi \;\mid\; \chi \wedge \chi \;\mid\; \mathit{onEvent}(\mu)\\
&\psi ::= \mathsf{var} \;\mid\; \psi\ \mathsf{arop}\ \psi \;\mid\; \mathsf{const} \;\mid\; \mathit{past}(\psi, \mathit{onEvent}(\mu), n) \;\mid\\
&\mathit{count}(\chi, K) \;\mid\; \mathit{count}(\chi, \mathit{onEvent}(\mu), K) \;\mid\; \mathsf{fun}(\psi, K) \;\mid\\
&\mathsf{fun}(\psi,\ \mathit{onEvent}(\mu), K) \;\mid\; \mathit{elapsed}(\mathit{onEvent}(\mu))\\
&\mathsf{op} ::= \mathtt{forall} \;\mid\; \mathtt{exists}\\
&\mathsf{relop} ::= < \;\mid\; \leq \;\mid\; = \;\mid\; \geq \;\mid\; >\\
&\mathsf{arop} ::= + \;\mid\; - \;\mid\; \times \;\mid\; \div\\
&\mathsf{fun} ::= \mathit{sum} \;\mid\; \mathit{avg} \;\mid\; \mathit{min} \;\mid\; \mathit{max}
\end{aligned}
$$

where id is an identifier, var is an internal or external variable, *onEvent* is an event predicate, *Becomes*, *Until*, *Between* and *Within* are temporal predicates, *count*, *elapsed*, *past*, and all the functions derivable from the non-terminal fun are temporal functions of the language. Parameter μ identifies an event: the *start* or the *end* of an *invoke* or *receive* activity, the receipt of a message by a *pick* or an *event handler*, or the execution of any other BPEL activity. K is a positive real number, n is a natural number and const is a constant.

The above syntax only defines the language's core constructs. The usual logical derivations are used to define other connectives and temporal operators (e.g., $\vee$, *Always*, *Eventually*, ...). Moreover, the strings derived from the non-terminal ϕ are called *formulae*; the strings derived from the non-terminal ψ are called *expressions*.

The formal semantics of ALBERT is provided in Appendix A.

3 A Bird-eye View of SAVVY-WS

This section illustrates the principles and the main design choices of SAVVY-WS. Its use is then illustrated in depth in the rest of this paper, which shows SAVVY-WS in action through two case studies.

SAVVY-WS's goal is to support the designers of composite services during the validation phase, which extends from design time to run time. SAVVY-WS assumes that service composition is achieved by means of the BPEL workflow language, which orchestrates the execution of external Web services.

Figure 2 summarizes the use of SAVVY-WS within the development process of BPEL service compositions. When a service composition is designed (step 1), SAVVY-WS assumes that the external services orchestrated by the workflow are only known through their specifications. The actual services that will be invoked at run time, and hence their implementation, may not be known at design time. The specification describes not only the syntactic contract of the service (i.e., the operations provided by the service, and the type of their input and output parameters), but also their expected effects, which include both functional and non-functional properties. Functional properties describe the behavioral contract of the service; non-functional properties describe its expected quality, such as its response time.

Specifying functional and non-functional properties only at the level of interfaces is required to support lifelong validation of dynamically evolvable compositions, which massively use late-binding mechanisms. Indeed, at design time a service refers to externally invoked services through their *required* interface. At run time, the service will resolve its bindings with external services that provide a matching interface, i.e., their *provided* interface conforms to the one used at design time.

The SAVVY-WS methodology supports the ALBERT language to specify required service interfaces. The language specifies the required interface in terms of logical formulae, called *assumed assertions* (AAs). Based on the AAs of all services invoked by the workflow, in turn, the composition may offer a service whose properties can also be specified via ALBERT formulae, called *guaranteed assertions* (GAs). Therefore, the second step of the SAVVY-WS-aware development process is to annotate the BPEL process with assumed and guaranteed assertions written in ALBERT (step 2 in Fig. 2).

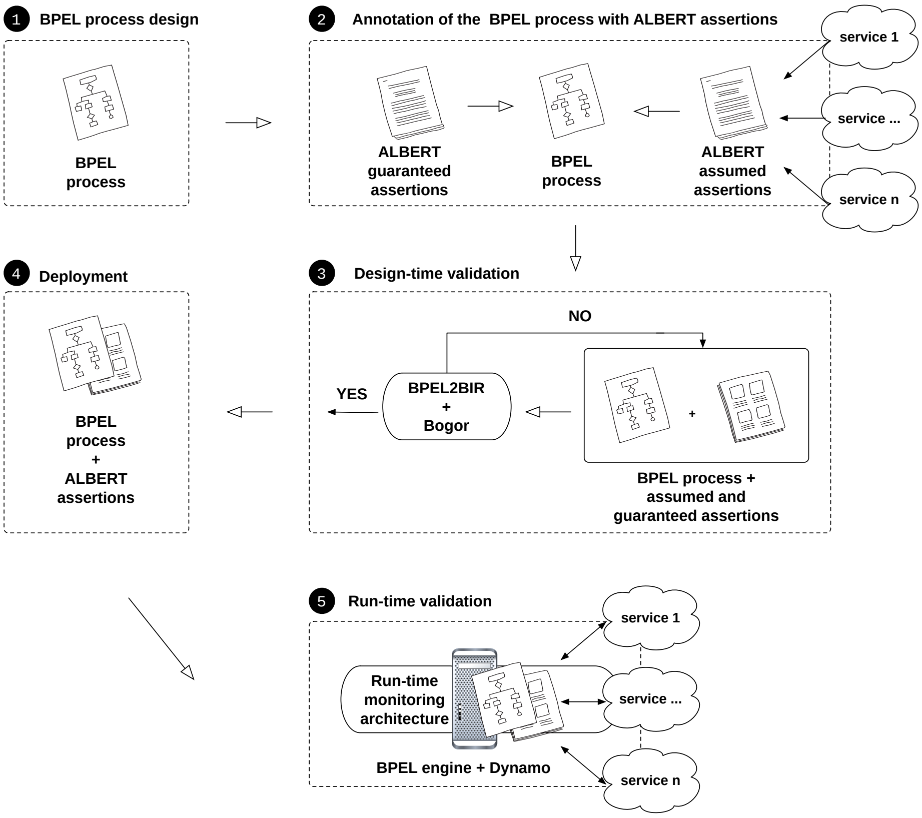

1 BPEL process design
BPEL process
2 Annotation of the BPEL process with ALBERT assertions
ALBERT guaranteed assertions
BPEL process
ALBERT assumed assertions
service 1
service ...
service n
3 Design-time validation
NO
YES
BPEL2BIR + Bogor
+
BPEL process + assumed and guaranteed assertions
4 Deployment
BPEL process + ALBERT assertions
5 Run-time validation
Run-time monitoring architecture
BPEL engine + Dynamo
service 1
service ...
service n

Fig. 2. SAVVY-WS-aware development process

The SAVVY-WS methodology is supported at design time by a formal verification tool (Bpel2Bir) that is used to check (step 3 in Fig. 2) that a composite service delivers its expected functionality and meets the required quality of service (both specified in ALBERT as GAs), under the assumption that the external services used in the composition fulfill their required interfaces (specified in ALBERT as AAs). The SAVVY-WS verification tool is based on the Bogor model checker [17].

Design-time verification does not prevent errors from occurring at run time. In fact, there is no guarantee that a service implementation eventually fulfills the contract promised through its provided interface. The service provider may either be malicious, by offering a service with an inferior experienced quality of service and/or a wrong functionality to increase its revenue on the service provision, or it might change the service implementation as part of its standard maintenance process: in this case, a service that worked properly might be changed in a new version that violates its previous contract.

Furthermore, during design-time verification, it is not possible to model the behavior of the underlying distributed infrastructure, which plays an important role in the provision of networked services. Although service providers' specifications could take into account, to some extent, the role of the distributed infrastructure, it is virtually impossible to foresee all possible conditions of the infrastructure components (e.g., network links) at design time.

To solve these problems, SAVVY-WS supports continuous verification by transforming —when a BPEL process is deployed on a BPEL execution engine (step 4 in Fig. 2)— ALBERT formulae into run-time assertions that are monitored (step 5 in Fig. 2) by Dynamo —our monitoring framework embedded within the BPEL engine— to check for possible deviations from the correct behavior verified at design time. If a deviation is caught, suitable compensation policies and recovery actions should be activated.

4 Running Examples

In this section, we describe the two running examples used in the rest of this paper to illustrate our lifelong validation methodology.

The first example is inspired by one of the scenarios developed in the context of the EU IST project SENSORIA [18]. We considered the *On Road Assistance* scenario, which takes place in an automotive domain, where a SOA interconnects (the devices running on) a car, service centers providing facilities like car repair, towing and car rental, and other actors. As will be described in Sect. 5.1, this example is used to show how to express (and validate) in ALBERT properties related to the timeliness of events.

The second example is inspired by a similar one described in [19, 20] and it illustrates a BPEL process realizing a *Car Rental Agency* service. It interacts with a *Car Broker Service*, which controls the operations of the branch; with a *User Interaction Service*, through which customers can make car rental requests; with a *Car Information Service*, which maintains a database of cars availability and allocates cars to customers; with a *Car Parking Sensor Service*, which exposes as a Web service the sensor that senses cars as they are driven in or out of the car parking of the branch. As will be illustrated in Sect. 5.2, this example will be used to show how to express ALBERT properties about sequences of events.

4.1 Example 1: *On Road Assistance*

The *On Road Assistance* process (depicted in Fig. 3), is supposed to run on an embedded module in the car and is executed after a breakdown, when the car becomes not driveable.

The *Diagnostic System* sends a message with diagnostic data and the driver's profile (which contains credit card data, the allowed amount for a security deposit payment, and preferences for selecting assistance services) to the workflow, which

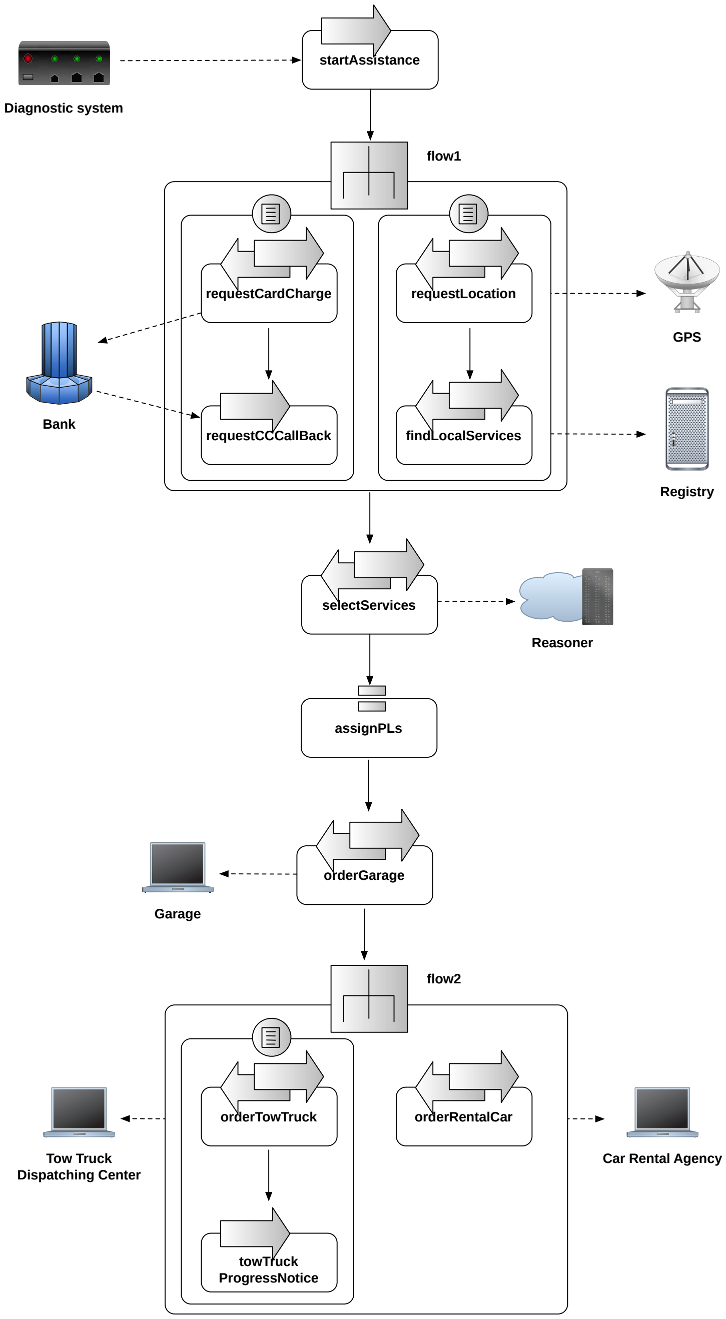


Fig. 3. The *On Road Assistance* BPEL process

starts by executing the `startAssistance` *receive* activity. Then, it starts a *flow* (named `flow1`) containing two parallel *sequence*s of activities.

In one *sequence*, the process first requests the *Bank* service to charge the driver's credit card with a security deposit payment, by invoking the operation `requestCardCharge` and passing the credit card data and the amount of the payment. Then, it waits for the asynchronous reply of the *Bank*, modeled by the `requestCCCallBack` *receive* activity.

In the other parallel *sequence*, the process first asks the *GPS* service —which represents a Web service interface for the GPS device installed on the car— to provide the position of the car (`requestLocation` *invoke* activity). The returned location is then used to query (`findLocalServices` *invoke* activity) a *Registry* to discover appropriate services close to the area where the car pulled out. The *Registry* service will return a sequence of triples —each of which contains a suitable combination of locally available services providing car repair shops, car rental, and tow trucking— stored in the `foundServices` process variable.

Subsequently, this variable is used as an input parameter in the `selectServices` operation of the *Reasoner* service, which is supposed to select the best available service triple matching the driver's preferences, and to store the selected services' endpoint references in the `bestServices` process variable. After assigning (`assignPLs` *assign* activity) the endpoint references to *partner link*s corresponding to the *Garage*, *Car Rental Agency* and *Tow Truck Dispatching Center* services, the process first sets an appointment with the garage, by sending to it the car diagnostic data (`orderGarage` *invoke* activity). The garage acknowledges the appointment by sending back the actual location of the repair shop.

Afterwards, the process starts a *flow* (named `flow2`) with three activities. Two activities are grouped in a *sequence*, where the process first contacts the towing service dispatching center (`orderTowTruck` *invoke* activity), and then it waits for an acknowledgment message `ack` confirming that a tow truck is in proximity of the car; this message is consumed by the `towTruckProgressNotice` *receive* activity.

The other activity is executed in parallel to the *sequence* mentioned above, and is used to contact the car rental agency (`orderRentalCar` *invoke* activity). In both *invoke* activities of `flow2`, the garage location is sent as an input parameter, representing the coordinates where the car is to be towed to and where the rental car is to be delivered.

To keep the example simple, we assume that at least one service triple is retrieved after invoking the *Registry*, and that the selected garage, towing service, and car rental agency can cope with the received requests.

4.2 Example 2: *Car Rental Agency*

The *Car Rental Agency* process (sketched in Fig. 4) is supposed to run on the information system of a local branch of a car rental company.

The process starts as soon as it receives a message from the *Car Broker Service* (`startRental` *receive* activity). Then, the process enters an infinite loop:

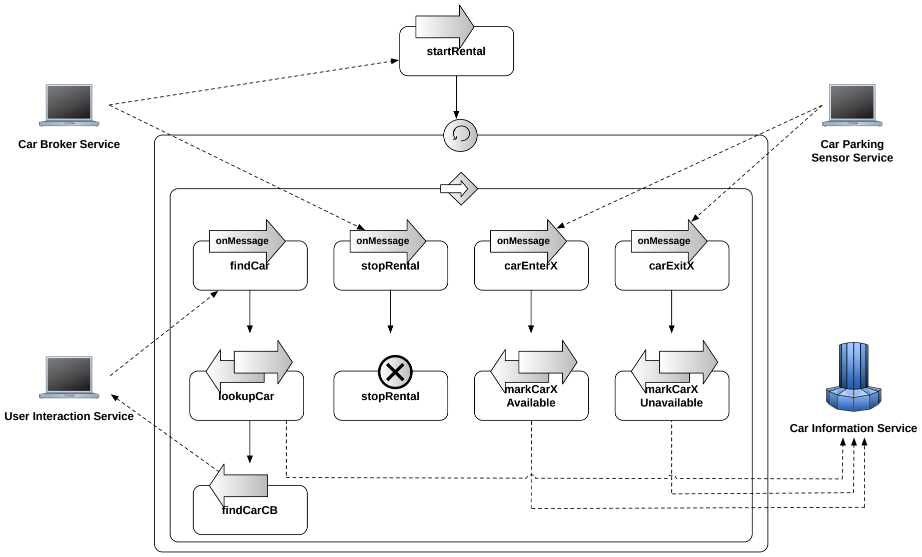


Fig. 4. *Car Rental Agency* BPEL process

every iteration is a *pick* activity that suspends the execution and waits for one of the following four messages:

- `findCar`. A customer asks to rent a car and provides her preferences (e.g., the car model). Then, the process checks the availability of a car that matches customer's preferences by invoking the `lookupCar` operation on the *Car Information Service*. The result of this operation, which can be either a negative answer or an identifier corresponding to the digital key to access the car, is returned to the customer with the `findCarCB` *reply* activity.
- `carEnterX` and `carExitX`. These two messages are sent out by the *Car Parking Sensor Service* when a car enters (respectively, exits) the car parking. The process reacts to this information by updating the cars database, invoking, respectively, the `markCarAvailableX` and `markCarUnavailableX` operations on the *Car Information Service*. Actually, the `X` in the name of each message or operation is a placeholder for a unique id associated with a car; therefore, if `A` is a car id, the actual messages associated with it have the form `carEnterA` and `carExitA`, and the corresponding operations are named `markCarAvailableA` and `markCarUnAvailableA`.
- `stopRental`. The *Car Broker Service* stops the operations of the local branch, making the process terminate.

To keep the example simple, we assume that the local branch where the *Car Rental Agency* process is run is the only one accessing the *Car Information Service* and that cars in the car parking can be only rented through the local car rental branch.

5 Specifying Services with ALBERT

The ALBERT language defines formulae that specify *invariant* assertions for a BPEL process. Two kinds of assertions can be specified using ALBERT:

- *assumed assertions (AAs)*, which define the properties that partner services are required to fulfill when interacting with the BPEL process;
- *guaranteed assertions (GAs)*, which define the properties that the composite service should satisfy, assuming that external services operate as specified.

Both kinds of assertions allow for stating functional and non-functional properties of services. As an example, an AA that should hold after the execution (as a post-condition) of an *invoke* activity `Act` on an external service S can be written in the following form:

$$\mathit{onEvent}(\texttt{end\_Act}) \rightarrow \texttt{\$myVar=EDS::getData()/var}$$

The antecedent of the formula contains the *onEvent* predicate, which is used to identify a specific point in the execution of the workflow. This point is represented by its argument: in this case, the keyword `end` denotes the point right after the end of the execution of activity `Act`. The consequent of the formula states that the value of the internal variable `myVar` of the process (the variable returned after invoking service S) must be equal to the value obtained by accessing an external data source (the *EDS* Web service endpoint), invoking the `getData` operation on it and retrieving the `var` part from the return message.

It is also possible to express non-functional AAs, such as latency in a service response. The following ALBERT formula specifies that the duration of an *invoke* activity `Act` should not exceed 5 time units:

$$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 5)$$

As in the previous formula, the antecedent identifies a certain point in the execution of the process where the consequent should then hold to make the assertion evaluate to true. This point is represented by the event corresponding to the `start` of activity `Act`. The consequent contains the *Within* operator, which evaluates to true if its first argument evaluates to true within the temporal bound expressed by the second argument; otherwise it evaluates to false. In other words, the consequent states that the event corresponding to the `end` of activity `Act` should occur within 5 time units from the current instant, which is the time instant in which the antecedent of the formula is true. We leave the choice of the most suitable timing granularity to the verification engineer, who can then properly convert the informal system requirements to formal, real-time specifications [21].

ALBERT can also be used to express GAs. For example, one may state an upper bound to the duration of a certain sequence of activities, which includes external service invocations, performed by a composite BPEL workflow in response to a user's input request.

As said above, ALBERT can be used to specify both AAs and GAs for BPEL processes. However, when defining AAs, formulae should only refer to the BPEL activities that are responsible for interacting with external services. Typically, AAs express properties that must hold after the workflow has completed an interaction with an external service. Hereafter we list a few specification templates which proved to be useful to express AAs in practical cases. In the templates, μ is an event identifying the start or the end of an *invoke* or *receive* activity, the reception of a message by a *pick*, or an *event handler*; ϕ and χ are ALBERT formulae; ψ and ψ' are ALBERT expressions; n is a natural number which is used to retrieve a certain value upon the nth-last occurrence of event μ in the past; K is a positive real number denoting time distances; fun is a placeholder for any function (e.g., average, sum, minimum, maximum) that can be applied to sets of numerical values.

- $\mathit{onEvent}(\mu) \rightarrow \phi$: it allows for checking property ϕ only in the states preceding or following an interaction with an external service;
- $\mathit{past}(\psi',\ \mathit{onEvent}(\mu),\ n) =\ \psi \rightarrow \phi$: it allows for checking property ϕ on the basis of past interactions of the workflow with the external world;
- $\mathit{Becomes}(\mathit{count}(\chi,\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain number of specific events;
- $\mathit{Becomes}(\mathsf{fun}(\psi',\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain value of an aggregate function.

5.1 Specifying the *On Road Assistance* Process

Hereafter we provide some properties of the *On Road Assistance* process: each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

- BankResponseTime

  After requesting to charge the credit card, the *Bank* will reply within 4 minutes when a low-cost communication channel is used, and it will reply within 2 minutes if a high-cost communication channel is used. In ALBERT this AA can be expressed as follows:

  $$\begin{aligned}&\mathit{onEvent}(\texttt{end\_requestCardCharge}) \rightarrow\\ &\qquad(\texttt{VCG::getConnection()/cost=`low'}\wedge\\ &\qquad\ \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 4) \vee\\ &\qquad(\texttt{VCG::getConnection()/cost=`high'}\wedge\\ &\qquad\ \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 2))\end{aligned}$$

  where VCG is the Web service interface for the local vehicle communication gateway, providing contextual information on the communications channels currently in use within the car. VCG::getConnection()/cost represents an external variable retrieved by invoking the getConnection operation on the VCG service and accessing the cost part of the returned message.

- AllButBankServicesResponseTime
  The interactions with all external services but the *Bank*, namely *GPS*, *Registry*, *Reasoner*, *Garage*, *Tow Truck Dispatching Center* and *Car Rental Agency* will last at most 2 minutes. This AA is expressed as a conjunction of formulae, each of which follows the pattern:

  $$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 2)$$

  where Act ranges over the names of the *invoke* activities interacting with the external services listed above.
- AvailableServicesDistance
  The *Registry* will return services whose distance from the place where the car pulled out is less than 50 miles. This AA can be expressed as follows:

  $$\begin{aligned}&\mathit{onEvent}(\texttt{end\_findLocalServices}) \rightarrow\\ &\qquad(\texttt{forall t in \$foundServices/[*] ;}\\ &\qquad\quad(\texttt{forall s in \$foundServices/t/[*] ;}\\ &\qquad\quad\texttt{s/distance} < 50))\end{aligned}$$

  where foundServices contains a sequence of triples, where elements contain a distance message part.
- TowTruckServiceTimeliness
  The *Tow Truck Dispatching Center* service selected by the *Reasoner* will provide assistance within 50 minutes from the service request. This AA can be expressed as follows:

  $$\mathit{onEvent}(\texttt{end\_selectServices}) \rightarrow (\texttt{\$bestServices/towing/ETA} \leq 50)$$

  where the ETA message part represents the maximum time bound guaranteed by a service to provide assistance.
- TowTruckArrival
  The time interval between the end of the order of a tow truck and the arrival of the ack message (notifying that the tow truck is in proximity of the car) is bounded by the ETA of the *Tow Truck Dispatching Center* service, that is 50 minutes. This AA can be expressed as follows:

  $$\begin{aligned}&\mathit{onEvent}(\texttt{end\_OrderTowTruck}) \rightarrow\\ &\quad\mathit{Within}(\ \mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 50)\end{aligned}$$

- AssistanceTimeliness
  The tow truck that will be requested will be in proximity of the car within 60 minutes after the credit card is charged. This property must be guaranteed to the user by the *On Road Assistance* process. It is a GA, whose validity is

(rather trivially) assured at design time by the AllButBankServicesResponseTime, the TowTruckServiceTimeliness and the TowTruckArrival AAs, and by the structure of the process. The property can be expressed as follows:

$$\mathit{onEvent}(\texttt{end\_requestCCCallBack}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 60)$$

5.2 Specifying the *Car Rental Agency* Process

Hereafter we provide some properties of the *Car Rental Agency* process. As we did in the previous section, each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

- ParkingInOut
  Between two events signaling that a car exits the car parking, an event signaling the entrance for the same car must occur. This AA can be expressed[4] as a conjunction of formulae, each of which follows the pattern:

  $$\mathit{onEvent}(\texttt{carExitX}) \rightarrow \mathit{Until}(\neg \mathit{onEvent}(\texttt{carExitX}), \mathit{onEvent}(\texttt{carEnterX}))$$

  where `X` ranges over the identifiers of the cars available in the local rental branch. Moreover, this formula can be combined, using a logical AND, with a similar constraint that refers to the `carEnterX` message.
- CISUpdate
  If a car is marked as available in the *Car Information Service*, and the same car is not marked as unavailable until a `lookupCar` operation for that car is invoked, then the `lookupCar` operation should not return a negative answer. This AA on the behavior of the *Car Information Service* can be expressed[5] as a conjunction of formulae, each of which follows the pattern:

  $$\begin{aligned}&(\mathit{onEvent}(\texttt{end\_markAvailableX}) \wedge \\ &\quad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{end\_markUnavailableX}), \\ &\quad\quad \mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X})) \\ &\rightarrow \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X} \wedge \\ &\quad \mathit{Eventually}((\mathit{onEvent}(\texttt{end\_lookupCar}) \wedge \texttt{\$queryResult/res!="no"})))\end{aligned}$$

  where `X` ranges over the identifiers of the cars available in the local rental branch, `carInfo` is the input variable of the `lookupCar` operation, whose output variable is `queryResult`.

[4] The semantics of the *Until* operator described in Appendix A guarantees that its first argument will not be evaluated in the current state.

[5] A more complete specification should also include that two `lookupCar` operations for the same car could not happen at the same time. However, this is guaranteed by the structure of the workflow.

– RentCar
  If a car enters in the car parking, and the same car does not exit until a customer requests it for renting, then this request should not return a negative answer. This is a GA, whose validity is (rather trivially) assured at design time by the ParkingInOut and CISUpdate AAs, and by the structure of the process. The property can be expressed as a conjunction of formulae, each of which follows the pattern:

$$\begin{aligned}
&(\mathit{onEvent}(\texttt{carEnterX}) \land \\
&\qquad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{carExitX}), \\
&\qquad\qquad \mathit{onEvent}(\texttt{start\_findCar}) \land \texttt{\$carInfo/id=X})) \\
&\to \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_findCar}) \land \texttt{\$carInfo/id=X} \land \\
&\mathit{Eventually}((\mathit{onEvent}(\texttt{start\_findCarCB}) \land \texttt{\$queryResult/res!="no"})))
\end{aligned}$$

  where `X` ranges over the identifiers of the cars available in the local rental branch, `carInfo` is the input variable of the `findCar` message, `queryResult` is the variable returned to the *User Interaction Service* by the `findCarCB` *reply* activity.

6 Design-time Verification

Our design-time verification phase is based on model checking. We developed Bpel2Bir, a tool that translates a BPEL process and its ALBERT properties into a model suitable for the verification with the Bogor model checker [17]. In the rest of this section, we illustrate, with the help of some code snippets, how the two running examples and their ALBERT properties are translated into BIR (Bogor's input language).

6.1 Example 1: Model Checking the *On Road Assistance* Process

A BPEL process is mapped onto a BIR **system** composed of threads that model the main control flow of the process and its *flow* activities.

Data types are defined by using an intuitive mapping between WSDL messages/XML Schema types and BIR primitive/record types. In this mapping, XML schema simple types (e.g., `xsd:int`, `xsd:boolean`) correspond to their equivalent ones in BIR (e.g., **int** and **boolean**). Moreover, the mapping also supports some XML schema facets, such as restrictions on values over integer domains (e.g., `minInclusive`) and `enumeration`, which is translated into an enumeration type. For example, the message that is sent by the *Diagnostic System* to the process, contains diagnostic data and the driver's profile (which includes credit card data, the allowed amount for the security deposit payment and preferences for selecting assistance services). This complex type can be modeled as follows, using a combination of record types in BIR:

```
enum TDiagnosticData {dd1, dd2}
enum TCustomerPreference {cp1, cp2}
enum TCreditCard {cc_c1, cc_c2}

record TStartMsg {
  TDiagnosticData diagData;
  TCreditCard ccData;
  int (1,10) deposit;
  TCustomerPreference cpData;
}
```

where we assume, based on the WSDL specification associated with the BPEL process, that the amount for the security deposit payment is an integer value between 1 and 10 and that dd1, dd2, cp1, cp2, cc_c1 and cc_c2 are enumeration values.

The input variables of *receive* activities and the output variables of *invoke* activities, whose values result from interactions with external services, can be modeled using non-deterministic assignments. For example, the `startAssistance` *receive* activity can be modeled as follows:

```
TStartMsg startMsg;
// other code
startMsg := new TStartMsg;
choose
  when <true> do startMsg.diagData:=TDiagnosticData.dd1;
  when <true> do startMsg.diagData:=TDiagnosticData.dd2;
end
// same pattern for generating credit card data
// and customer's preferences
choose
  when <true> do startMsg.deposit :=1;
  when <true> do startMsg.deposit :=2;
         ...
  when <true> do startMsg.deposit :=9;
  when <true> do startMsg.deposit :=10;
end
```

Activities nested within a *flow* are translated into separated threads. In our example (see Fig. 3), `flow1` contains two *sequence* activities; `flow2` contains a *sequence* and an *invoke* activity. For each of these activities, we declare a corresponding global **tid** (thread id) variable:

```
tid flow1_sequence1_tid;
tid flow1_sequence2_tid;
tid flow2_sequence1_tid;
tid flow2_invoke1_tid;
```

For each activity in the *flow* we declare a thread, named after the corresponding **tid** variable. This thread contains the code that models the execution of the corresponding activity. For example, the thread corresponding to the *sequence* that includes requestCardCharge and requestCCCallBack activities, has the following structure:

```
thread flow1_sequence1() {
// code modeling requestCardCharge
// code modeling requestCCCallBack
exit;
}
```

Finally, the actual execution of a *flow* is translated into the invocation of a helper function launchAndWaitFlow$_i$, which creates and starts a thread for each activity in the flow, and returns to the caller only when all the launched threads terminate. This function has the following form (in the case of flow1):

```
function launchAndWaitFlow1() {
  boolean temp0;
  loc loc0: do {
    flow1_sequence1_tid := start flow1_sequence1();
    flow1_sequence2_tid := start flow1_sequence2();
  } goto loc1;

  loc loc1: do {
    temp0 := threadTerminated(flow1_sequence1_tid)
             && threadTerminated(flow1_sequence2_tid);
  } goto loc2;

  loc loc2: when temp0 do{} return;
            when !temp0 do{} goto loc1;
}
```

The assignPL activity is not translated since it only updates the partner link references of the process and thus it does not change the state of the process.

Once the basic model of the BPEL process has been created, it can be then enriched by exploiting *assumed assertions*. AAs can provide a better abstraction of the values deriving from the interaction with external services and they can also express constraints on the timeliness of the activities involving external services.

For example, property TowTruckServiceTimeliness represents a constraint on the value of variable bestServices. This means that we can restrict the range of the values that can be non-deterministically assigned to that variable, when modeling the output variable of the selectServices activity. This is shown in the following code snippet:

```
choose
  when <true> do bestServices.towing.ETA :=1;
  when <true> do bestServices.towing.ETA :=2;
          ...
  when <true> do bestServices.towing.ETA :=49;
  when <true> do bestServices.towing.ETA :=50;
end
```

The next example shows how AAs can be used to define time constraints for modeling either the execution time of, or the time elapsed between BPEL activities. The adopted technique is based on previous work on model checking temporal metric specifications [22]. We insert a code block that randomly generates the duration of the activity within a certain interval, bounded by the value specified in an AA. For *flow* activities, the time consumed by the *flow* is the maximum time spent along all paths. By focusing on `flow2` (see Fig. 3) of our example and using properties AllButBankServiceResponseTime and TowTruckArrival, we get the following code:

```
int (0,52) flow2_sequence1_clock;
int (0,2) flow2_invoke1_clock;
//other code
thread flow2_sequence1() {
  // code modeling orderTowTruck
  choose
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 1;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 2;
        end
  //code modeling TowTruckProgressNotice
  choose
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 1;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 2;
                 ...
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 49;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 50;
  end
}

thread flow2_invoke1() {
  // code modeling orderRentalCar
  choose
    when <true> do flow2_invoke1_clock :=
         flow2_invoke1_clock + 1;
    when <true> do flow2_invoke1_clock :=
         flow2_invoke1_clock + 2;
  end
}
//other code
active thread MAIN {
  //other code
  launchAndWaitFlow2();
  if flow2_sequence1_clock >= flow2_invoke1_clock do
```

```
    assistanceTimeliness_clock :=
      assistanceTimeliness_clock + flow2_sequence1_clock;
  else do
    assistanceTimeliness_clock :=
      assistanceTimeliness_clock + flow2_invoke1_clock;
  end
  //other code
}
```

The first two lines of the previous code snippet represent the declarations of local counters associated with the activities included in the *flow* (in this case a *sequence* and an *invoke*). The domain of these variables is bounded by the duration of each activity, as expressed in an AA; for structured activities (e.g., a *sequence*), we take as upper-bound the sum of the durations of all nested activities.

Each of these counters is then non-deterministically incremented in the body of the thread that simulates the execution of an activity. After the end of the execution of the *flow*, we take the maximum time spent along all paths and assign it to a global counter, associated with the process (starTowTruckProgressNotice_clock in our example).

The last step before performing the verification of the model is represented by translating into BIR the GA we want to verify. In our example, we want to prove that the time elapsed between the end of activity requestCCCallBack and the start of activity TowTruckProgressNotice is less than 60 time units (minutes). To achieve this, we declare a (global) clock that keeps track of the elapsed time; this is the global variable assistanceTimeliness_clock introduced above. Moreover, we need a boolean flag that will be set to true right after the end of activity requestCCCallBack, to enable access to the global counter. The AssistanceTimeliness property can then be translated into a simple BIR assertion:

```
assert(assistanceTimeliness_clock <= 60);
```

Before emitting the actual BIR code, Bpel2Bir performs a static analysis on the flow graph of the BIR program to detect data variables (i.e., the ones associated with inbound messages activities like *receive* and *invoke*) that are not used in the computation of the process. If such variables exist, we perform an optimization that removes them and the corresponding generative code blocks from the BIR model, to reduce the size of the model itself.

The verification of the (optimized) model of the process has been performed on a Intel Core 2 Duo 2.1 GHz processor running Apple Mac OS X 10.5.3 and Bogor ver. 1.2. The verification of property AssistanceTimeliness took 175s; the model had 708002 states and 2178206 transitions.

6.2 Example 2: model checking the *Car Rental Agency* Process

The basic structure of the *Car Rental Agency* process contains a *while* loop, with a *pick* activity that waits, at each iteration, for one of the messages described in Sect. 4.2. This structure is modeled in the following BIR code snippet:

```
active thread MAIN() {
  boolean operating; operating := true;
  while operating do choose
      when <true> do //code modeling findCar
        //code modeling lookupCar
      when <true> do //code modeling carEnterX
        //code modeling markCarAvailableX
      when <true> do //code modeling carExitX
        //code modeling markCarUnavailableX
      when <true> do //code modeling stopRental
        operating := false;
      end
  end
}
```

We translated the *pick* activity into a **choose** statement, which models the occurrence of one of the events waited for by the *pick* activity. In this way, we automatically model the mutual exclusion for the occurrence of the events and the non-determinism in selecting among events that occurred simultaneously. Variable `operating` is a boolean flag that keeps the process receiving messages from external services; it is set to false when a `stopRental` message arrives, making the *while* activity and then the process terminate. We do not translate the `findCarCB` *reply* activity, since it represents an outgoing communication with an external service, which does not modify the state of the process.

To model the `carInfo` variable, which is associated with the arrival of message `findCar` and the `lookupCar` operation, we declare the `TCarInfoID` **enum** type[6] and the corresponding variable in the BIR model, as shown in the following code snippet:

```
enum TCarInfoID {c1, c2, c3}
TCarInfoID carInfo;
```

This variable is assigned a value by means of a **choose** statement, when the `findCar` message is selected by the outer **choose** statement, which models the enclosing *pick* activity. Since there are two nested **choose** statements, it is possible to optimize the generated code by flattening and producing only one **choose** statement, with a number of alternatives equal to the combination of the incoming messages and their input variables. The resulting code follows this structure:

```
choose
  when <...> do
    carInfo := TCarInfoID.c1;
    //other code modeling findCar C1
  when <...> do
    carInfo := TCarInfoID.c2;
    //other code modeling findCar C2
    //other code modeling the other alternatives
```

[6] To keep the example simple, we assume that there are only three cars available for renting, to which the three identifiers declared in the enumeration correspond.

```
  when <...> do
    //other code modeling carExit C3
end
```

The AAs defined for this process can help improve and enrich the BIR model. For example, property ParkingInOut adds some constraints on when the arrival of a carExitX (or a carEnterX) message can be "simulated" by the **choose** statement. The intuitive meaning of property ParkingInOut (a carExitX message cannot be received if the last message received was another carExitX) is translated into a guard for the **when** statement. The guard consist of a boolean variable named after the message name (e.g., carEnter_c1, carExit_c1). The boolean flag is then assigned a proper value when the corresponding event occurs: e.g., carExit_c1 is assigned the true value when the alternative of the **choose** statement equivalent to the corresponding event is selected. The following code snippet clarifies how these boolean variables are dealt with:

```
boolean carEnter_c1;
boolean carExit_c1;
...
when <!carEnter_c1> do  //code modeling carEnterC1
  carEnter_c1 := true;
  carExit_c1  := false;
  //other code
```

As the reader may notice, variable carEnter_c1 is true whenever carExit_c1 is false, and vice versa. However, they are kept distinct as the translator cannot deduce this relation by simply analyzing the logical predicates of a formula. A further optimization step includes additional input from the user, when the relation between two variables/predicates is provided to the translator.

Property CISUpdate makes the translator emit the definition of similar boolean flags corresponding to the execution of activities markAvailableX, markUnavailableX and lookupCar. Moreover, it also defines how to generate the return value corresponding to the invocation of the lookupCar operation. The following code snippet exemplifies this behavior:

```
when <true> do  //code modeling findCar c3
  carInfo := TCarInfoID.c3;
  // code modeling lookupCar c3
  if markAvailable_c3 && !markUnavailable_c3 do
    queryResult_res_DiffNo := true;
  else do
    queryResult_res_DiffNo := false;
  end
```

where queryResult_res_DiffNo is the boolean variable corresponding to the predicate $queryResult/res!="no".

Last, the RentCar GA can be translated into an **assert** expression, guarded by a logical predicate corresponding to the antecedent of the formula, as shown in the following code snippet:

```
if carEnter_c3 && !carExit_c3 do
  assert(queryResult_res_DiffNo == true);
end
```

Since the `findCarCB`, as said before, is not modeled, this assertion is placed right after the code modeling the arrival of the `findCar` message.

The verification of the model of the process in Figure 4 has been performed using the same configuration detailed in the previous section. It took 24ms to verify property RentCar; the model had 85 states and 106 transitions. By comparing the order of magnitude of the experimental data of the two examples, the reader will observe how the use of explicit time bounds, as in the *On Road Assistance* example, may increase the complexity of a model.

7 Run-time Monitoring

In SAVVY-WS, service compositions are validated at run time by monitoring AAs and GAs via Dynamo, our dynamic monitoring framework.

Monitoring rules specify the directives for the monitoring framework; each of them comprises two main parts: a set of *Monitoring Parameters* and a *Monitoring Property* expressed in ALBERT. *Monitoring Parameters* allow our approach to be flexible and adjustable with respect to the context of execution. They are meta-level information used at run time to decide whether a rule should be monitored or not. We provide three parameters:

- `priority`, which defines a simple "notion" of importance among rules, ranging over five levels of priority. When a rule is about to be evaluated, its priority is compared with a threshold value, set by the owner of the process; the rule is taken into account if its priority is less than or equal to the threshold value. By dynamically changing the threshold value we can dynamically set the intensity of probing.
- `validity`, which defines time constraints on *when* a rule should be considered. Constraints can be specified in the form of either a time window or a periodicity. The former defines time-frames within which monitoring is performed; when outside of this frame, any new monitoring activities are ignored. The latter specify how often a rule should be monitored; accepted values are durations and dates, e.g., "`3D`", meaning every 3 days, or "`10/05`" meaning every May 10th.
- `trusted providers`, which defines a list of service providers who do not need to be monitored.

Figure 5 presents the technologies we used in the implementation, as well as how the various components interact among themselves. We have chosen to adopt ActiveBPEL [23], an open-source BPEL server implementation, as our Execution Engine, and to extend it with monitoring capabilities by using aspect-oriented programming (AOP) [24]. The Data Manager represents the advice code that is weaved into the engine. When the engine initiates a new process instance, the Data Manager loads all that process' ALBERT formulae from the Formulae

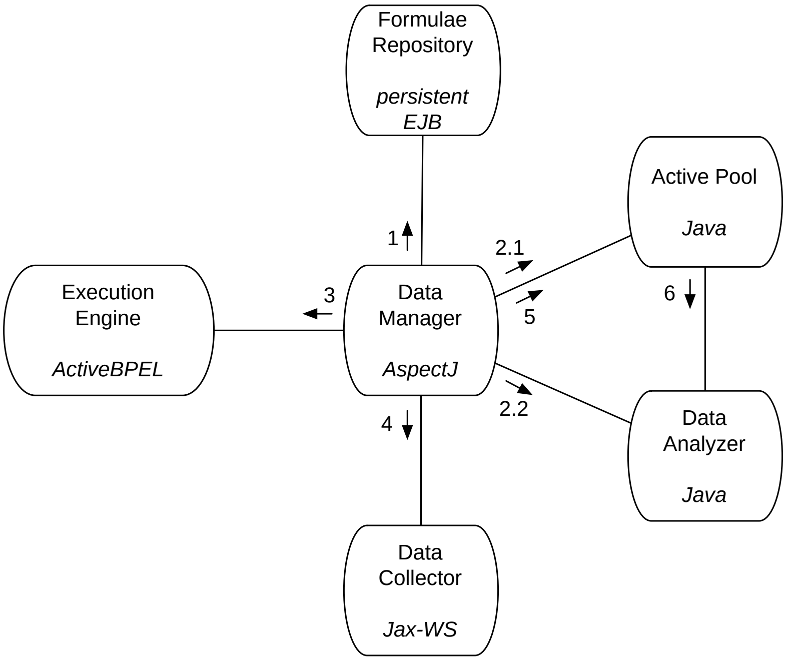


Fig. 5. Monitoring framework

Repository (step 1), and uses them to configure and activate both the Active Pool and the Data Analyzer (steps 2.1 and 2.2). The former is responsible for maintaining (bounded) historical sequences of process states, while the latter is the actual component responsible for the analysis.

The Data Manager's main task stops the process every time a new state needs to be collected for monitoring. This is facilitated by the fact that it has free access to the internals of the executing process. Once all the needed ALBERT internal variables are collected (step 3), the process is allowed to continue its execution. Notice that ALBERT formulae may also refer to external data, which do not belong to the business logic itself. In this case the data collected from the process need to be completed with data retrieved from external sources (e.g., context information), and this is achieved through special-purpose Data Collectors (step 4). Once collected, the internal and the external data make up a single process state. At this point the state is time-stamped, labeled with the location in the process from which the data were collected, and sent to the Active Pool (step 5), which stores it. Every time the Active Pool receives a new state it updates its sequences to only include the minimum amount of states required to verify all the formulae. The sequences are then used by the Data Analyzer to check the formulae (step 6).

The evaluation of ALBERT formulae that contain only references to the present state and/or to the past history (i.e., formulae that do not contain

Until, *Between*, or *Within* operators) is straightforward. On the other hand, the evaluation of formulae that contain *Until*, *Between*, or *Within* operators depends on the values the variables will assume in future states. From a theoretical point of view, this could be expressed by referring to the well-known correspondence between Linear Temporal Logic and Alternating Automata [25]. From an implementation point of view, the Data Analyzer relies on additional evaluation threads for evaluating each subformula containing one of the three aforementioned temporal operators.

Run-time monitoring inevitably introduces a performance overhead. Indeed we need to temporarily stop the executing process at each BPEL activity to collect the internal variables that constitute a new state. Meanwhile, if any external variables are needed they are collected after the process resumes its execution. Therefore, the overhead is due to two main factors: the time it takes the AOP advice to stop the process and activate internal data collection, and the collection time itself. Exhaustive tests have allowed us to quantify the former in less than 30 milliseconds. This is the time it takes the advice code to obtain the list of internal variables it needs to collect. The actual collection time, on the other hand, depends on the number of internal variables we need to collect. Once again our tests have shown that, on average, it takes 2 milliseconds per internal variable. This is due to the fact that the AOP advice code has direct access to the process' state in memory, and that ActiveBPEL provides an API method for doing just that.

More details on how the different components of this architecture work for monitoring ALBERT properties are given in [12]. Instead, in the next two sections, we will focus on the Data Analyzer, by describing how it evaluates the properties of our running examples.

7.1 Example 1: Monitoring the *On Road Assistance* Process

The first property we consider is BankResponseTime. When the `requestCardCharge` activity is executed, the Data Manager detects, by accessing the Formulae Repository, that a property is associated with the end of the execution of the activity. Right after the activity completes, the Data Analyzer starts evaluating the consequent of the formula.

Since the root operator of the consequent is a logical OR, the Data Analyzer evaluates the left operand first, i.e., the first conjunction. The left conjunct is a reference to an external variable: the Data Analyzer asks the Data Collector to invoke the operation `getConnection` on the Web service `VCG` and then it checks the value of the `cost` part of the return message. If the value is equal to `'low'`, the Data Analyzer evaluates the other operand of the logical AND, that is the *Within* subformula.

The evaluation of such a formula cannot be completed in the current state, thus the Data Analyzer spawns a new thread to evaluate the formula in future states of the process execution. This thread checks for the truth value of its formula argument, i.e., for the occurrence of the event (notified by the Active Pool) corresponding to the start of the execution of activity `requestCCCallBack`, while keeping track of the progress of a timer, bounded by the second argument of the *Within* formula. If the formula associated with the *Within* operator becomes true before the timer reaches its upper bound, the thread returns true, otherwise it returns false.

Since the evaluation of logical AND and OR operators is short-circuited, if the evaluation of the external variable returned by the Data Collector returns false, the second operand (i.e., the *Within* formula) is not evaluated, making the Data Analyzer start evaluating the other operand of the logical OR, following a similar pattern (accessing the external variable, spawning a thread for checking the *Within* formula, checking the value returned by this thread). Similarly, if the first operand of the logical OR evaluates to true, the second operand is not evaluated.

Property AllButBankServiceResponseTime can be monitored in a similar way, but without the need for accessing external variables through the Data Collector. When one of the activities bounded to the `Act` placeholder is started, the Data Analyzer spawns a new thread, waiting for the end of the corresponding activity, within the time bound.

AvailableServicesDistance and TowTruckServiceTimeliness are two examples of properties that can be evaluated immediately. As a matter of fact, as soon as the execution of the activity listed in the antecedent of the formula finishes, the Data Analyzer retrieves the current state of the process from the Active Pool, and it evaluates the variables referenced in the formula.

Finally, the monitoring of properties TowTruckArrival and AssistanceTimeliness, follows the evaluation patterns seen above. Both formulae include a *Within* subformula, which requires an additional thread for the evaluation.

7.2 Example 2: Monitoring the *Car Rental Agency* Process

Monitoring of property ParkingInOut is triggered by the arrival of a `carExitX` message, intercepted during the execution of a *pick* activity. Right after the arrival of the message, the Data Analyzer evaluates the consequent of the formula, whose operator is an *Until*. Such a formula cannot be evaluated in the current state, thus the Data Analyzer spawns a new evaluation thread. This thread receives notifications from the Active Pool about new process states being available. When a notification arrives, the thread evaluates the second subformula of the *Until* operator, i.e., it waits for the occurrence of the event `carEnterX`. If this subformula evaluates to false, the thread evaluates (in a similar way) the first subformula of the *Until* operator. If this subformula is also false, the evaluation thread terminates by returning false. Otherwise, the thread continues to evaluate the *Until* formula in future states.

In property CISUpdate, the evaluation of the antecedent of the formula requires to spawn a new thread, since it contains an *Until* subformula. First, the Data Analyzer checks for the end of the execution of activity `markAvailableX` and then waits for the thread evaluating the *Until* subformula to terminate. This thread evaluates the formula in a similar way as described above, in the case of the ParkingInOut formula. Once the evaluating thread terminates, the overall antecedent of the formula, i.e., the logical AND, is evaluated. The consequent of the formula is thus evaluated only when the logical AND in the antecedent is true; since it contains the *Eventually* operator, its evaluation requires a new thread to be spawned. This thread checks for the occurrence of the event corresponding to the start of activity `lookupCar`; then, it spawns a new thread —since there is a second, nested, *Eventually* operator— that then waits for the end of the execution of activity `lookupCar` and checks for the value of variable `queryResult`.

Property RentCar is evaluated in a similar way, since the formula follows the same pattern of the previous one.

8 Related Work

The work presented in [19] is similar to SAVVY-WS, since it also proposes a lifelong validation framework for service compositions. The approach is based on the Event Calculus of Kowalski and Sergot [26], which is used to model and reason about the set of events generated by the execution of a business process. At design time the control flow of a process is checked for livelocks and deadlocks, while at run time it is checked if the sequence of generated events matches a certain desired behavior. The main difference with SAVVY-WS is the lack of support for data-aware properties.

Many other approaches investigated by current research tackle isolated aspects related to the main issue of engineering dependable service compositions. Design-time validation is addressed, for example, in [27], where the interaction between BPEL processes is modeled as a conversation and then verified using the SPIN model checker. In [28], design specifications (in the form of Message Sequence Charts) and implementations (in the form of BPEL processes) are translated into the Finite State Process notation and checked with the Labelled Transition System Analyzer. Besides finite state automata and process algebras, Petri Nets represent another computational model for static verification of service compositions. They are used to model both BPEL [29] and BPMN [30] processes; in both cases, the verification focuses on detecting unreachable activities and deadlocks.

Other approaches focus on run-time validation of service compositions, considering either the behavior, as in [31, 20], or the non-functional aspects [32–34], or both [35].

Design- and run-time validation activities are related to the language that is used to specify the properties that are to be validated. Besides more traditional approaches based on assertion languages like WSCoL [36], or languages for defining service-level agreements (SLAs), such as WSLA [37], WS-Agreement [38] and SLAng [39], a third trend is based on languages for defining policies, such as WS-Policy [40]. An extension of WS-Policy, called WS-Policy4MASC, is defined in [35] to support monitor and adaptation of composite web services. Even though it is not specifically bound to a validation framework, the StPowla approach [41] aims at supporting policy-driven business modeling for SOAs. StPowla is a workflow-based approach that attaches to each task of a workflow modeling a business process, a policy that expresses functional and non-functional requirements and business constraints on the execution of the task.

9 Conclusion

The paper provided a tutorial introduction to the SAVVY-WS methodology, which supports the development and operation of Web service compositions by means of a lifelong validation process. SAVVY-WS's goal is to enable the development of flexible SOAs, where the bindings to external services may change dynamically, but still control that the composition fulfills the expected functional and non-functional properties. This allows the flexibility of dynamic change to be constrained by correctness properties that are checked during design of the architecture and then monitored at run time to ensure their continuous validity.

SAVVY-WS has been implemented and validated in the case of Web services compositions implemented in the BPEL workflow language. The approach, however, has a more general scope.

It can be generalized to different composition languages and to other implementations of SOAs, which do not use Web service technologies. We have described our long-term research vision in [42]: leveraging the experience gained while working on SAVVY-WS, we want to develop SAVVY, a complete methodology for lifelong validation of dynamically evolvable software service compositions. The ultimate goal of SAVVY is to integrate specification, analysis and verification techniques, in a technology-independent manner, supported by a rich set of tools.

Acknowledgements. Part of this work has been supported by the EU project "PLASTIC" (contract number IST 026995) and by the EU project "S-Cube", funded within FP7/2007-2013 under Objective 1.2 "Services and Software Architectures, Infrastructures and Engineering".

References

1. Baresi, L., Di Nitto, E., Ghezzi, C.: Towards Open-World Software. IEEE Computer **39** (2006) 36–43
2. ICSOC: International Conference on Service-Oriented Computing series. `http://www.icsoc.org` (2003–2008)
3. SeCSE Project: Description of Work. `http://secse.eng.it/` (2004)
4. PLASTIC Project: Description of Work. `http://www.ist-plastic.org` (2005)
5. S-CUBE: S-CUBE network. `http://www.s-cube-network.eu/` (2008)
6. NESSI: Networked European Software and Services Initiative. `http://www.nessi-europe.com` (2005)
7. Papazoglou, M.: Web Services: Principles and Technology. Prentice Hall (2008)
8. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web services: Concepts, Architectures, and Applications . Springer (2003)

9. The SeCSE Team: Designing and deploying service-centric systems: The SeCSE way. In: Proceedings of Service-Oriented Computing: a look at the Inside (SOC@Inside'07), workshop co-located with ICSOC 2007. (2007)
10. ART DECO Project: Description of Work. `http://artdeco.elet.polimi.it/Artdeco` (2005)
11. DISCoRSO project: Project vision. `http://www.discorso.eng.it` (2006)
12. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Softw. **1**(6) (2007) 219–232
13. Ghezzi, C., Inverardi, P., Montangero, C.: Dynamically evolvable dependable software: from oxymoron to reality. In Degano, P., De Nicola, R., Meseguer, J., eds.: Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Volume 5065 of Lecture Notes in Computer Science. Springer (2008) 330–353
14. Bianculli, D., Ghezzi, C.: SAVVY-WS at a glance: supporting verifiable dynamic service compositions. In: Proceedings of the 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS 2008), IEEE Computer Society Press (2008) to appear.
15. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1 (2003)
16. OMG: Business process modeling notation, v.1.1. `http://www.omg.org/spec/BPMN/1.1/PDF` (2008) OMG Available Specification.
17. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building your own software model checker using the Bogor extensible model checking framework. In: Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 148–152
18. Wirsing, M., Carizzoni, G., Gilmore, S., Gonczy, L., Koch, N., Mayer, P., Palasciano, C.: SENSORIA: Software engineering for service-oriented overlay computers. `http://www.sensoria-ist.eu/files/whitePaper.pdf` (2007)
19. Rouached, M., Perrin, O., Godart, C.: Towards formal verification of web service composition. In: Proceedings of the 4th International Conference on Business Process Management (BPM 2006). Volume 4102 of Lecture Notes in Computer Science., Springer (2006) 257–273
20. Mahbub, K., Spanoudakis, G.: A framework for requirements monitoring of service based systems. In: Proceedings of the 2nd international conference on Service-Oriented computing (ICSOC '04), ACM Press (2004) 84–93
21. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software engineering (ICSE '05), New York, NY, USA, ACM (2005) 372–381
22. Bianculli, D., Spoletini, P., Morzenti, A., Pradella, M., San Pietro, P.: Model checking temporal metric specification with Trio2Promela. In: Proceedings of International Symposium on Fundamentals of Software Engineering (FSEN 2007), Teheran, Iran. Volume 4767 of Lecture Notes in Computer Science., Springer (2007) 388–395
23. Active Endpoints: ActiveBPEL Engine Architecture. `http://www.activebpel.org/docs/architecture.html` (2006)
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241 of Lecture Notes in Computer Science., Springer (1997) 220–242

25. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata. Volume 1043 of Lecture Notes in Computer Science., Springer (1996) 238–266
26. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gen. Comput. **4**(1) (1986) 67–95
27. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th International Conference on World Wide Web (WWW '04), ACM Press (2004) 621–630
28. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003) , IEEE Computer Society (2003) 152–163
29. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2-3) (2007) 162–198
30. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri Nets. `http://eprints.qut.edu.au/archive/00007115/` (2007)
31. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: Proceedings of the International Conference on Web Services (ICWS '06), Washington, DC, USA, IEEE Computer Society (2006) 63–71
32. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web (WWW'08), ACM (2008) 815–824
33. Raimondi, F., Skene, J., , Emmerich, W.: Efficient monitoring of web service SLAs. In: Proceedings of the 16th International Symposium on the Foundations of Software Engineering (SIGSOFT 2008 - FSE 16), ACM (2008) to appear.
34. Sahai, A., Machiraju, V., Sayal, M., Jin, L.J., Casati, F.: Automated SLA monitoring for web services. In: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '02). Volume 2506 of Lecture Notes in Computer Science., Springer (2002) 28–41
35. Erradi, A., Maheshwari, P., Tosic, V.: WS-Policy based monitoring of composite web services. In: Proceedings of the 5th European Conference on Web Services (ECOWS '07), IEEE Computer Society (2007) 99–108
36. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC '05). Volume 3826 of Lecture Notes in Computer Science., Springer (2005) 269–282
37. Keller, A., Ludwig, H.: The WSLA framework: specifying and monitoring service level agreement for web services. Journal of Network and System Management **11**(1) (2003)
38. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement). `http://www.ogf.org/documents/GFD.107.pdf` (2007)
39. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of the 26th International Conference on Software Engineering (ICSE '04), IEEE Computer Society (2004) 179–188
40. W3C Web Services Policy Working Group: WS-Policy 1.5. `http://www.w3.org/2002/ws/policy/` (2007)

41. Gorton, S., Montangero, C., Reiff-Marganiec, S., Semini, L.: StPowla: SOA, Policies and Workflows. In: Proceedings of the 3rd International Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition (WESOA 2007). (2007)
42. Bianculli, D., Ghezzi, C.: Towards a methodology for lifelong validation of service compositions. In: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments (SDSOA 2008), co-located with ICSE 2008, Leipzig, Germany, ACM (2008) 7–12

A ALBERT Formal Semantics

The formal semantics of the ALBERT language [12] is defined over a *timed state word*, an infinite sequence of states $s = s_1,\ s_2,\ \ldots,$ where a state s_i is a triple $(V_i,\ I_i,\ t_i)$. V_i is a set of $\langle\psi, value\rangle$ pairs with ψ being an expression that appears in a formula, I_i is a location of the process[7] and t_i is a time-stamp. States can therefore be considered snapshots of the process.

The arithmetic (arop) and mathematical (fun) expressions behave as expected. Function $past(\psi, onEvent(\mu),\ n)$ returns the value of ψ, calculated in the nth state in the past in which $onEvent(\mu)$ was true. Function $count(\chi,\ K)$ returns the number of states, in the last K time instances, in which χ was true, while its overloaded version $count(\chi, onEvent(\mu),\ K)$ behaves similarly but only considers states in which $onEvent(\mu)$ was also true. Finally, function $elapsed(onEvent(\mu))$ returns the time elapsed from the last state in which $onEvent(\mu)$ was true.

For all timed words s, for all $i \in \mathbb{N}$, the *satisfaction relation* $\models$ is defined as follows:

- $s,\ i \models \psi$ relop ψ' iff $eval(\psi, s_i)$ relop $eval(\psi', s_i)$
- $s,\ i \models \neg\phi$ iff $s,\ i \not\models \phi$
- $s,\ i \models \phi \wedge \xi$ iff $s,\ i \models \phi$ and $s,\ i \models \xi$
- $s,\ i \models onEvent(\mu)$ i ff
  - if μ is a start event, $\mu \in I_{i+1}$,
  - otherwise, $\mu \in I_i$
- $s,\ i \models Becomes(\chi)$ i ff$i > 0$ and $s,\ i \models \chi$ and $s,\ i-1 \not\models \chi$
- $s,\ i \models Until(\phi,\xi\ )$ i ff$\exists j > i \mid s,\ j \models \xi$ and $\forall k$, if $i < k < j$ then $s,\ k \models \phi$
- $s,\ i \models Between(\phi, \xi, K)$ i ff$\exists j \geq i \mid s,\ j \models \phi$ and $\forall l$ if $i \leq l < j$ then $s,\ l \not\models \phi$ and $\exists h\ \ \mid t_h \leq t_j + K,\ t_{h+1} > t_j + K$ and $s,\ h \models \xi$
- $s,\ i \models Within(\phi, K)$ i ff$\exists j \geq i \mid t_j - t_i \leq K$ and $s,\ j \models \phi$

where function *eval* takes an ALBERT expression ψ and a state in the timed state word s_i and returns the value of ψ in s_i.

[7] A location is defined as a set of labels of BPEL activities; in the case of a *flow* activity, it contains, for each parallel branch of the *flow*, the last activity executed in that branch.

Exploiting Assumption-Based Verification for the Adaptation of Service-Based Applications

Andreas Gehlert
University of Duisburg-Essen
Schützenbahn 70
45117 Essen, Germany
andreas.gehlert@sse.uni-due.de

Antonio Bucchiarone
FBK-IRST
via Sommarive 18
38100 Trento, Italy
bucchiarone@fbk.eu

Raman Kazhamiakin
FBK-IRST
via Sommarive 18
38100 Trento, Italy
raman@fbk.eu

Andreas Metzger
University of Duisburg-Essen
Schützenbahn 70
45117 Essen, Germany
andreas.metzger@sse.uni-due.de

Marco Pistore
FBK-IRST
via Sommarive 18
38100 Trento, Italy
pistore@fbk.eu

Klaus Pohl
University of Duisburg-Essen
Schützenbahn 70
45117 Essen, Germany
klaus.pohl@sse.uni-due.de

ABSTRACT

Service-based applications (SBAs) need to operate in a highly dynamic world, in which their constituent services could fail or become unavailable. Monitoring is typically used to identify such failures and, if needed, to trigger an adaptation of the SBA to compensate for those failures.

However, existing monitoring approaches exhibit several limitations: (1) Monitoring individual services can uncover failures of services. Yet, it remains open whether those individual failures lead to a violation of the SBA's requirements, which would necessitate an adaptation. (2) Monitoring the SBA can uncover requirements deviations. However, it will not provide information about the failures leading to this deviation, which constitutes important information needed for the adaptation activities. Even a combination of (1) and (2) is limited. For instance, a requirements deviation will only be identified after it has occurred, e. g., after the execution of the whole SBA, which then in case of failures might require costly compensation actions.

In this paper we introduce an approach that addresses those limitations by augmenting monitoring techniques for individual services with formal verification techniques. The approach explicitly encodes assumptions that the constituent services of an SBA will perform as expected. Based on those assumptions, formal verification is used to assess whether the SBA requirements are satisfied and whether a violation of those assumptions during run-time leads to a violation of the SBA requirements. Thereby, our approach allows for (a) pro-actively deciding whether the SBA requirements will be violated based on monitored failures, and (b) identifying the specific root cause for the violated requirements.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitor*

Keywords

Service-oriented computing, verification, monitoring

1. INTRODUCTION

1.1 Need for Adaptation

Service Based Applications (SBAs) are composed of services provided by service providers that are often different from the company that is operating the SBA [28]. Such a distribution of computational resources and software comes with the advantage to flexibly use any service available on the network and, therefore, to adapt the SBA to new business situations by, for instance, exchanging one service for another. This flexibility, however, comes at the cost of losing tight control over the SBA, as the SBA owner cannot control the provisioning, execution, management and evolution of externally provided services [28]. This means that the SBA designer must rely on the ability of the service providers to meet the expected functionality and quality of those services (encoded, for instance, as service-level agreements).

Once the SBA is put into operation, those expectations may—intentionally—or unintentionally – be violated; for instance, a service might fail. The operator of the SBA must not only recognise these violations but also decide whether those violations mean that the overall SBA requirements will no longer be met. In such a case an adaptation of the SBA can become necessary.

1.2 Limitations of Monitoring

Currently, monitoring is used to trigger the adaptation of a service-based application. However, existing monitoring techniques—as detailed below—exhibit several limitations

SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

which impact on taking adaptation decisions. Failing to make those decisions may lead to unnecessary or harmful adaptations [20].

Monitoring individual services: It is possible to monitor specific events and elements of the SBA, such as monitoring the constituent services [15]. Such approaches recognise whether a service delivers the expected functionality or quality. However, it is unclear whether this violation of the contract eventually leads to a violation of the SBA's requirements. Without this information we cannot decide whether the SBA should be adapted or not. Assume, for instance, that a service takes $1s$ longer than expected. It may be the case that the service is part of a parallel control flow in the service composition and that such a delay does not have any impact on the performance of the parallel control flow and thus the overall quality of the SBA.

Monitoring service compositions / SBAs: The requirements to the whole service composition may be monitored [3]. In this way it is possible to check whether the composition behaves as required. However, in this case, the identification of the source of the requirements violation is not trivial. Assume, for instance that a service composition takes $30s$ longer than expected to terminate. "Debugging" as an additional step would then be needed to determine, which service(s) caused that delay. It is important to know the cause of the problem to compensate for it by adapting the system; e. g., one could replace the service that caused the delay.

Combined efforts: Even a combination of the above two techniques has limitations. Indeed, in case of complex SBAs, a variety of events and violations may occur. How to debug and identify a specific cause of the requirements violation in order to trigger proper adaptation actions remains an open problem in such a case. Additionally, even with the combined approach we can only identify a problem of the service composition when this is identified by monitoring. This especially means that it is not possible to "predict" whether a violation of a service contract (detected by monitoring individual services) may eventually lead to a violation of the requirements of the service composition.

1.3 Contribution of the Paper

In this paper we present an approach that aims at addressing the above limitations. More specifically, our approach is able to detect run-time problems and violations of SBA's requirements and to identify specific root causes for those problems in order to determine appropriate adaptation actions. To achieve this, our approach augments monitoring techniques (to detect service failures) with formal verification techniques (to determine requirements violations). The central idea of our approach is to observe specific properties—assumptions—that (1) are explicitly related to the requirements and (2) characterize the constituent services of the SBA. Thereby, our approach allows (a) verifying whether a problem can lead to a violation of requirements, and (b) tracing the violation to its root cause, which facilitates adaptation.

The remainder of the paper is structured as follows: In Section 2 we introduce the notion of assumptions and how those are exploited in our approach. This is followed by a scenario to illustrate the approach in Section 3. We detail the individual steps of our approach in Section 4 and discuss related work in Section 5. Section 6 discusses the results and possible future work.

2. ASSUMPTION-BASED VERIFICATION

The concept of "assumption" is well understood in software and requirements engineering. While requirements can be influenced and realised by the designer building the system, assumptions can only be affected by agents in the system's environment [19, 30]. Thus, neither the designer nor the system itself have any influence on the violation or validity of assumptions [31]. Provided that a system fulfils its requirements under a given set of assumptions, a violation of those assumptions may lead to a situation, in which the software system does not provide the expected quality anymore and, therefore, deviates from its requirements[9].

In the case of service-based applications, assumptions may characterize the constituent services (e. g., their interfaces, QoS parameters, etc.) and/or the context (e. g., infrastructure, business, context, user, etc.). Once the assumptions are established (e. g., expected QoS is agreed through a contract such as a service-level agreement), the designer of the SBA expects that those assumptions are valid during the design- and run-time phases of the system (e. g., the contract will not be violated by the provider).

To decide whether a violation of an assumption (e. g., local deviation from contracts of one or more services) leads to a violation of the SBA requirements, we need an analysis technique. For the purpose of this paper, we propose to exploit verification techniques in two ways. First, a verification of the system at design time allows us to prove that the design corresponds to the requirements provided that the assumptions hold. Second, if the re-verification executed at run-time reports a violation of the corresponding requirements, we conclude that the observed violation of an assumption will lead to problems in the SBA. As the assumptions characterize a specific element (e. g., a specific service), an appropriate adaptation action may be triggered (e. g., substitute a service).

The presented approach benefits from (1) the possibility to identify the specific source of the problem and trace it to the critical element of the domain that should be handled by adaptation; (2) only specific assumptions need to be monitored to identify potential problems (thus, no need to monitor the whole requirements and compositions); and (3) in case of long-running applications the run-time monitoring and analysis results may be exploited to trigger adaptation activities pro-actively, before the failure actually takes place.

3. ILLUSTRATIVE SCENARIO

We use the following example to illustrate our approach. The company "Green Transport IT Solutions" wants to support passengers in cities to use the available public transport systems and develops a service-based application, which computes travel routes for helping users navigate from one place in a city to another. This application is deployed on a mobile device and uses different services for each city depending on the location of the device. The system implements the following workflow (cf. Figure 1): after the user has entered his or her destination, the system locates the user and computes the quickest route to the next bus stop. After this step, the system calculates the public transport route to the destination bus stop. Lastly, the shortest footpath from this bus stop to the final destination is calculated.

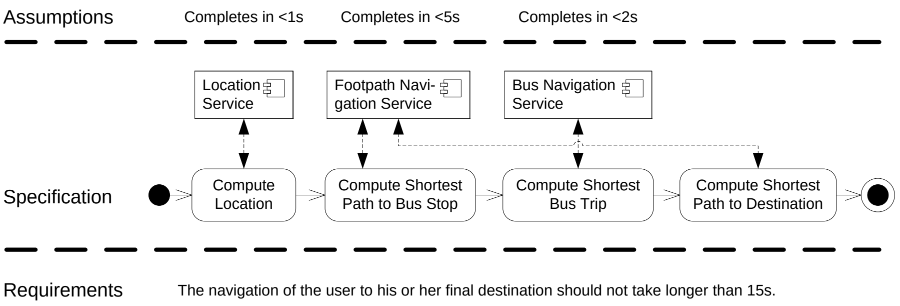

Assumptions
Completes in <1s
Completes in <5s
Completes in <2s
Location Service
Footpath Navigation Service
Bus Navigation Service
Specification
Compute Location
Compute Shortest Path to Bus Stop
Compute Shortest Bus Trip
Compute Shortest Path to Destination
Requirements
The navigation of the user to his or her final destination should not take longer than 15s.

Figure 1: Assumptions, Requirements and Specifications of a Mobile Navigation Scenario

Figure 1 presents a non-functional requirement of the SBA at the bottom, the specification of the workflow in the middle and the assumptions regarding execution time at the top. The linear workflow of four steps should be executed within $15s$. The workflow invokes a location service once, a footpath navigation service twice and a bus navigation service once. According to the services' individual SLAs (i. e., service assumptions) their time to complete the request are at most $1s$ for the location service, $5s$ for the footpath navigation service and $2s$ for the bus navigation service. Furthermore, it is assumed that the service interaction times may be neglected (context assumption). Based on these assumptions, the initial verification performed during run-time proves that even in the worst case (upper bound for response time), the process terminates in less than $13s$ and, therefore, the requirement of the SBA is fulfilled.

Assume now the following two situations during run-time:

- *The location service takes* $3s$. While, its SLA is violated, an adaptation is not needed as the process takes now exactly $15s$. Monitoring the whole requirement or following our approach would thus have avoided an unnecessary adaptation.

- *The location service takes* $3s$ *and the the footpath navigation service takes* $7s$. If only the SBA would be monitored in that case, it would not be possible to understand which of the services have failed and which of the services should be adapted. As an example, if a different location service was chosen, the overall time of the service composition would be reduced to $16s$ and, therefore, the requirement would still be violated. In our approach, instead, if we re-verify the system, we can detect that the SBA requirement is violated regardless whether the assumption about the location service is violated or not. Therefore, in this case, it is necessary to exchange the footpath navigation service.

Besides the timed properties above, it is also possible to consider, monitor, and analyze functional aspects of the service composition. For example, the services may have complex interaction protocols consisting of several interactions, such as "make a navigation request" or "refine the destination". The SBA requirement in this case may express the deadlock freeness, while the assumption may express the necessity of all possible service implementations to follow the same interaction protocol. Indeed, if a specific service implementation does not follow the same protocol, run-time re-verification may check whether the new protocol is still compatible with the requirement.

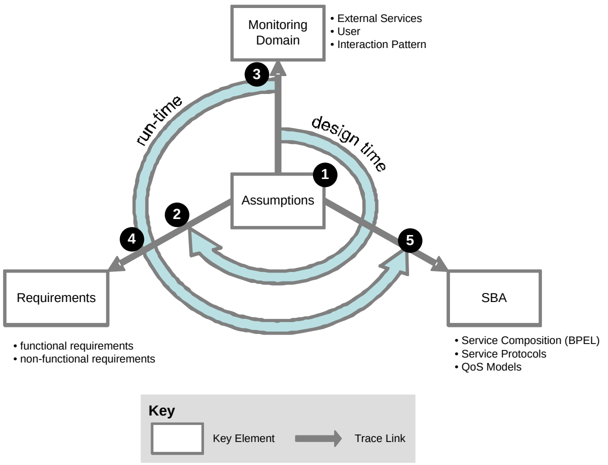


Figure 2: High-Level View on the Proposed Approach

4. DETAILED APPROACH

Figure 2 depicts the main elements of our approach. As explained in the introduction the approach builds on a clear separation between requirements for the SBA and the assumptions under which it is supposed to operate. Furthermore, we distinguish between the system itself and its context (or domain). As the SBA's constituent services are provided by different service providers and as these providers control those services with regard to functionality and quality, they belong to the SBA's domain.

Our approach involves the following five steps (cf. numbers in Figure 2):

1. This first step addresses the design-time where the designer documents assumptions and requirements separately.

2. In the second step the designer verifies the system and deploys the system only if the system passes the verification step.

3. At run-time the assumptions are monitored.

4. A violated assumption triggers a run-time re-verification.

5. If the SBA does not pass the verification given the violated assumptions, an adaptation is triggered, because this means that the SBA requirements are violated.

Those steps are described in more detail in the following subsections. Each subsection contains two aspects. The first aspect describes the concepts of each step in more detail. The second aspect describes techniques that can be used to implement the concepts of that step and thus demonstrates how our approach can be realized based on existing techniques for monitoring and verification. The aim of the selected techniques is to demonstrate the feasibility of our concepts only. Thus, we do not claim that we have selected the most appropriate techniques for each phase and, therefore, do not provide a discussion of possible alternative techniques.

4.1 Step 1: Separating Requirements & Assumptions

During the design step of the SBA its requirements and assumptions are documented together with the SBA model. Assumptions may be defined for the user behaviour, for the device on which the SBA runs and for the services provided by service providers. Here, we only focus on assumptions for services.

The assumptions may be extracted in different ways. In particular, it is possible to exploit domain knowledge (e. g., the properties of the telecommunication infrastructure adopted in different cities), to rely on historical information obtained and continuously updated during the execution of previous versions of the SBA, etc. The specific methods for eliciting domain assumptions is outside of the scope of this paper and will be addressed in future work.

4.1.1 Modeling Assumptions

Concepts: Assumptions may address functional (e. g., the interfaces and the protocol used by the service) and quality aspects (e. g., response time or availability) of services. As we verify the system later on (see Section 4.2) we will need formal techniques to describe these functional and quality aspects.

During this step, the assumptions, requirements and the elements of the SBA's specification are related by two types of trace links: First, the link between assumptions and requirements allows tracing a violation of this assumption back to its requirement. This link ensures that we do not only know the violated assumption but also the consequence for the SBA with respect to the fulfilment of its requirements. Second, the link between the assumption and the specification allows us to trigger an adaptation of the relevant part of the SBA once a violation of an assumption is detected.

Implementation: We propose to produce a set of template-based documents, which describe assumptions verbally and formally as well as their relations to the requirements and SBA specification. Such templates allow providing a name, a natural language description, a type, a formal description, the related requirements and the affected SBA element(s) (cf. Table 3 for examples). The assumption is described by a short and unique name as well as a natural language description. In addition, we document the assumption's type. Here, we distinguish between assumptions on functional, behavioral and non-functional properties of services. The formal description of the assumption is needed for verification and to monitor it at run-time. Lastly, the assumption is related to a requirement (related requirement row) and to a specific SBA element, which was designed using this assumption (affected SBA element row).

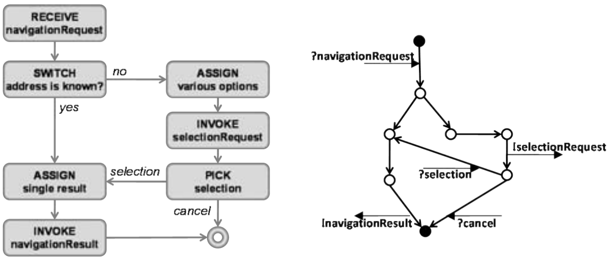


Figure 4: BPEL Protocol (left) and STS (right) of the Footpath Navigation Service

4.1.2 Modeling the Service-based Application

Concepts: A composite SBA may be defined using a language such as BPEL (Business Process Execution Language). The component services are represented by their interfaces (defined, e. g., in WSDL) that define a set of possible service operations, their parameters, data types, and binding protocol information.

Implementation: Formally, we represent the composed application and the protocol of the component services as a State Transition System (STS, [21]). A STS describes a dynamic system that can be in one of its possible states and can evolve to new states as a result of performing some actions. Actions are distinguished in input actions, which represent the reception of messages ("receive" and "onMessage" BPEL activities), output actions, which represent messages sent to external services ("invoke" and "reply"), and internal actions, modeling internal computations and decisions ("assign", "switch", etc.). The STS formalism may be extended to Timed Transition System [21] to capture time properties for representing assumptions on duration of activities, BPEL timeouts, etc.

An example of the abstract BPEL protocol and the corresponding STS of the Footpath Navigation Service is shown in Fig. 4. The protocol (left) starts when the navigation request is received. If the specified location is known the service returns an appropriate route (sends "navigationResult" message). Otherwise, the service prepares and sends to the requester the set of possible options with the "selectionRequest" operation. The requester may perform a selection (i. e., send the selection message) or cancel the procedure (sending the cancel message). In the former case the route is returned. In the latter case the procedure terminates. The protocol is reflected by the corresponding STS.

4.2 Step 2: Perform Design-Time Verification

Concepts: Once the system is designed, we need to formally verify that it satisfies its requirements under the specified assumptions. This formal verification is a necessary activity in our approach since this verification formally ensures that the deployed SBA corresponds to the requirements and assumptions documented in step 1. If this activity was not included in our approach and we detected a violation of an assumption at run-time (see Section 4.4), we would not know whether this violation is the source of the problem or whether the problem existed in the SBA in the first place.

Implementation: There are many SBA verification techniques available, which can be used for our purposes [25, 14, 22]. To verify for functional correctness and quality correctness the approach takes STSs and models in linear

Assumption:	Timing Assumptions
Description:	a1) The location service completes in at most 1s. a2) The footpath navigation service completes in at most 5s. a3) The bus navigation service completes in at most 2s.
Type:	Non-Functional
Formal Specification:	a1) $time(RECEIVED(locationResult)\ Since\ SEND(locationRequest)) < 1$ a2) $time(RECEIVED(footpathNavigationResult)\ Since\ SEND(footpathNavigationRequest)) < 5$ a3) $time(RECEIVED(busNavigationResult)\ Since\ SEND(busNavigationRequest)) < 2$
Related Requirement:	The system should be able to calculate the entire route to the destination in at most $15s$.
Affected SBA Element:	Location service, footpath navigation service, bus navigation service.

Figure 3: Documented timed assumptions

temporal logic (LTL) as input. Together with approaches, which translate property sequence charts into timed Büchi automata [2], the approach allows to bridge the gap between the design step and the analysis step.

To check whether the SBA satisfies its functional and non-functional requirements we use model checking techniques [14, 25, 26] that are able to check if the behavioural model of the SBA is conform to the given functional correctness properties representing the SBA requirements. We apply the approach proposed in [22] to verify that the SBA of our running example is correct. The approach takes as input the STSs representing the SBA and the services as well as the behavioral correctness properties defined in linear temporal logic (LTL). The properties may express for instance, deadlock freeness, ordering constraints on events, starvation, etc. It uses NuSMV [8] as a model-checker.

Instead of writing directly temporal properties, one can use the algorithm proposed in [2] that translates Property Sequence Charts (PSCs) into a Büchi automaton [7]. The notation of PSC [2] is very close to UML sequence diagrams, but has a formal underlying semantics suitable for verification purposes. After the verification of the behavior of the composition model at design-time, and in case a violation is detected at run-time, a counterexample (also in PSC-like form) that demonstrates the erroneous execution of the composition is provided.

In our scenario it is necessary to verify non-functional requirements, such as time properties. In particular, the global time constraint (execution time $\leq 15s$) should be satisfied. This requirement can only be satisfied if all the services participating in the application fulfill their own local timed constraints. To verify such time constraints, the approach proposed in [21] may be exploited. The approach extends the verification techniques presented above to take time properties and assumptions of service compositions into account. As input the approach takes the specification of the SBA and service behavior enriched with temporal aspects such as timeouts and constraints on activity durations, expressed as *Timed Transition Systems*. For specifying correctness properties that explicitly speak about time, the approach exploits the *duration calculus* (DC) notation [17]. Formally, the notation is close to the model of LTL and PSC; it enables the use of similar verification algorithms (the NuSMV model checker has been used for this purpose).

4.3 Step 3: Monitor Assumptions at Run-Time

Concepts: Upon the completion of the second step the system is deployed and used. To identify service faults, we need to monitor the services—more precisely we need to monitor the services w. r. t. the fulfilment of their service level agreements. Therefore, the assumptions (expressing the SLAs) should be mapped to monitoring rules, which are in turn used by a monitoring engine to monitor the SBA.

Implementation: For the assumptions monitoring purpose we adopt the integrated Astro/Dynamo framework and the corresponding monitoring language [6].

The choice is motivated by the capabilities and expressive power of the framework. First, the language permits specifying properties in a declarative way using the notations similar to those used for the verification (i. e., using temporal logic constructs similar to LTL and DC). In particular, it allows for capturing timing and statistical information about process activities. For example the property that the overall interaction with the Footpath Navigation Service should not exceed 1 second may be expressed using the following formula:

$$time(RECEIVED(navigationResult)$$
$$Since\ SEND(navigationRequest)) < 1$$

Second, the Astro/Dynamo monitoring framework allows expressing properties and measure/aggregate information over a single execution of a process instance as well as over a set of instances; these are important properties if assumptions refer to certain statistical information (e. g., certain QoS aspects, such as performance or availability). Third, the framework allows for measuring the properties in the SBA's context using external services and components as a pluggable sources of information. This is an important capability in order to evaluate context assumptions. Details on the realization of this capability can be found in [15].

Consequently, the timed assumptions, behavioral assumptions, and certain contextual assumptions may be observed using the integrated monitoring framework. As for protocol assumption monitoring, we rely on the approach presented in [29]. The underlying idea is that a monitor observes an interaction with the corresponding service and compares the sequence of messages with the one defined in the protocol. If wrong messages arrive, the protocol is violated, and the violation is reported by the monitoring framework.

4.4 Step 4: Run-Time Verification

Concepts: If a violation of an assumption is detected by monitoring, we need to determine whether this violation actually leads to a failure in the SBA. To evaluate such failures we use the verification techniques described in Section 4.2 at run-time. An adaptation is only necessary if the verification fails. As we have monitored the individual assumptions, we know the source of the problem, e. g. the assumption that was violated. Together with our traceability links (see again Figure 3), we can determine the affected requirement(s). Having analysed the impact of the service fault, the SBA operator can decide whether an adaptation is necessary or not. It is important to note here, that the adaptation can take effect before the SBA instance terminates, as the verification has determined the violation of the SBA requirements even before the whole workflow has been executed.

Implementation: We can use the same techniques as presented in Section 4.2. For example, if the protocol of the Navigation Service has changed (e. g., an error message is sent if the location specified by the user is unknown), it is possible to verify whether the whole composition is still deadlock-free. If the requirement is still satisfied, the adaptation is not needed.

In a similar way, from the re-verification of timed properties, it is possible to evaluate whether the time requirement is violated, and which of the violated assumptions is responsible for that.

4.5 Step 5: Triggering Adaptation

Concepts: If the requirement has found to be violated, the next activity is to identify the source of the problem and trigger appropriate adaptation actions. In our approach the identification of the problem source is straightforward, because any assumption is associated to the elements of the application model that rely upon those assumptions. This relation allows us to identify the component of the system that is subject to adaptation.

Implementation: A specific element of the SBA domain, such as constituent services and the context (e. g., connectivity, or user profiles) may be further associated with the appropriate adaptation actions. For example, the service protocol and SLAs may be associated with such actions as replacement of the service, re-design/re-compose the part of the orchestrating BPEL process that interacts with that service in order to accommodate to the new version, etc.

Note that the ability to precisely identify the critical elements of the domain allows us to identify adaptation actions that are the most appropriate in a given situation and, therefore, avoid redundant or harmful adaptations.

5. RELATED WORK

The closest relations to our approach may be found in the requirements engineering domain under the label requirements monitoring. Fickas and Feather for instance, describe in [12, 13] an approach similar to ours. In line with us, the authors argue, that assumptions should be identified during the requirements engineering phase of a software development project. The authors also argue that the system only operates correctly in case of valid assumptions. They conclude that assumption monitoring is beneficial to adapt the system (called remedial action in the papers) once a violation of an assumption is detected. However, Fickas and Feather's work is descriptive only, e. g. concrete techniques for monitoring assumptions and for linking them with the system to derive adaptation triggers are missing.

More recent work of Cohen et al. and Fickas et al. concentrate on the monitoring aspect. In the [9, 11] the authors describe a flexible method how to derive monitors based on requirements and assumptions, e. g. elicited and documented by the KAOS approach [10]. Because of the strong monitoring focus, the papers do not cover any aspect related to adaptation.

The idea of requirements monitoring is also used in the domain of Web services and service compositions, where a wide range of approaches have been proposed for monitoring services and service compositions. In particular, in [4, 5] the authors propose an approach for BPEL monitoring. The monitoring properties (functional and non-functional) are expressed as pre-post conditions on service invocations within the process specification. In [24] the authors also propose a framework for BPEL composition monitoring. These approaches, however, do not differentiate between assumptions and generic requirements, and therefore, cannot be directly applied for identifying the source of the problem and for triggering appropriate adaptations.

In [29] and [23] the authors exploit assumptions for automated service composition. In the former case a composed BPEL process is automatically built using assumptions on the protocols of the component services. Moreover, run-time monitoring matches the actual behaviors of the service composition against the assumptions expressed in the composition requirements, and report violations. In the latter case an OWL-S composition is constructed exploiting assumptions on SBA context expressed as logical constraints in the Semantic Web Rule Language (SWRL) [27]. The composition obtained in such a way satisfies the requirements (composition goals) if the assumptions hold. These approaches have as main goal the service composition using assumptions. While in the first case they present also a way to monitor the composition at run-time to find possible violations, the second presents only a way to compose services without monitoring them. Neither the first nor the second approach, however, address the issue of debugging the violation of the assumptions at run-time.

Regarding the approaches that aim at identifying the causes of the occurring deviations from requirements, in [1] authors propose a framework for Web Service orchestration, which employs diagnostic services to support a fine grained identification of the causes of exceptions (occurring during the execution of a composite service) and the consequent execution of effective exception handlers. This is achieved by defining a special infrastructure with Local Diagnoser services associated to each component service. These services generate diagnostic hypotheses over exceptions from the local point of view, while Global Diagnoser service aggregates these hypotheses to provide a global diagnosis of the occurred failure. With respect to our approach, this work focuses only on specific types of exceptions and faults; it does not evaluate the implication of these exceptions on the SBA requirements, and requires heavy instrumentation also on the side of the constituent services, which reduces the flexibility and dynamicity of the SBAs.

In [18] the authors propose a framework that exploits assumptions for the design and maintenance of software systems. At design time assumptions are defined over different modules of the system. The verification is exploited (1) to check that the component satisfies the assumptions and (2) to guarantee that the whole system behaves correctly (using "assume-guarantee" reasoning [16]). During software system evolution, if the software code changes, the assumptions are re-checked and the possible violation of the system correctness is reported or new assumptions are generated. Differently to our approach, this framework deals only with a very specific type of properties (e. g., program code assertions) and leaves open the problem of how the changes are monitored.

6. DISCUSSION AND PERSPECTIVES

In this paper we have demonstrated how monitoring techniques can be beneficially augmented with verification techniques to support the adaptation of service-based applications. The basic idea of our approach is to start from explicitly documented requirements and assumptions. Assumptions address functional and quality properties of third-party services (e. g., as documented in service-level agreements). A verification step at design time ensures that the SBA fulfils its requirements under specified assumptions. During run-time, monitoring the assumptions allows detecting violations (e. g., service failures). A violation of SBA's requirements can then be determined by re-verifying the SBA given the violated set of assumptions. If that verification fails, an adaptation, to compensate for the violation of the assumptions, may be triggered.

It is important to note that this paper does not discuss whether an adaptation of the SBA needs to be realised. We only provide adaptation triggers. Each adaptation trigger indicates that a requirement of the running SBA instance will be violated. Wether this violation is worth to be addressed by an adpation and how this adaptation should be realised is subject to further research.

Our approach exploits formal verification techniques. By doing so, we limit our approach to those requirements and assumptions, which can be formally expressed. In addition, the verification of complex systems may take considerable resources so that it may not be feasible to use these techniques at run-time. Both issues are subject to future work.

The current discussion of our approach is limited to the service composition layer—in particular we do not address the infrastructure layer of the service-based application. However, this infrastructure is important for any quality of service attribute related to time. Especially, if the execution times for individual services are low, the time needed for the communication between services need to be taken into account. The design of our approach does not take this communication time into account and is, therefore, based on the assumption that these communication times are minimal compared to the execution time of the SBA's services.

Our approach foresees a verification at run-time to determine whether a service failure may lead to a violation of the SBA's requirements. However, even if an assumption is violated, this might not lead to a requirements violation and thus the verification might not fail. To save computational resources at run-time it would, thus, be desirable to define a minimal set of assumptions such that each violation of an assumption will lead to a failure in the SBA and, thus, eliminates the costly verification step at run-time. In the future we are going to investigate whether this approach for various types of assumptions and models would be feasible.

Furthermore, we argued that due to the identification of the problematic part in the SBA, the adaptation could be better tailored to the failure situation and is, thus, more efficient. Since the approach is based on the general concept of assumptions, it should also be possible to extend it to other types of assumptions, e. g., assumptions about users, devices, locations and other context factors and, therefore, to trigger an adaptation based on factors, which are outside the SBA's boundaries. In future work we plan to substantiate both claims by investigating the interplay between our approach and current adaptation strategies.

Lastly, we argued that assumptions are engineered during the design step. In reality, however, assumptions may also be derived during the verification step. If a verification fails at design time, this failure may be due to the fact that some assumptions were missing. Consequently, it would be very interesting to understand the interplay between requirements engineering and verification in order to derive assumptions, which fit the need of both techniques.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

7. REFERENCES

[1] L. Ardissono, R. Furnari, A. Goy, G. Petrone, and M. Segnan. Fault tolerant web service orchestration by means of diagnosis. In V. Gruhn and F. Oquendo, editors, *EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2006.

[2] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Eng.*, 14(3):293–340, 2007.

[3] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *IEEE International Conference on Web Services (ICWS 2006)*, pages 63–71, 2006.

[4] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM.

[5] L. Baresi and S. Guinea. Towards dynamic monitoring of ws-bpel processes. In *ICSOC*, pages 269–282, 2005.

[6] L. Baresi, S. Guinea, M. Trainotti, and M. Pistore. Dynamo + ASTRO: An integrated approach for bpel monitoring. In *7th International Conference on Web Services (ICWS 2009)*, 2009.

[7] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science*, pages 1–1. Stanford Univ. Press, 1962.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[9] D. Cohen, M. S. Feather, K. Narayanaswamy, and S. S. Fickas. Automatic monitoring of software requirements. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 602–603, New York, NY, USA, 1997. ACM.

[10] A. Dardenne, A. V. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, pages 3–50, 1993.

[11] S. Fickas, T. Beauchamp, and N. A. R. Mamy. Monitoring requirements: A case study. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 299, Washington, DC, USA, 2002. IEEE Computer Society.

[12] S. Fickas and M. S. Feather. Requirements monitoring in distributed environments. In *SDNE '95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, page 93, Washington, DC, USA, 1995. IEEE Computer Society.

[13] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 140, Washington, DC, USA, 1995. IEEE Computer Society.

[14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03*, pages 152–161. IEEE, 2003.

[15] C. Ghezzi and S. Guinea. Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services*, pages 237–264, 2007.

[16] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1991.

[17] M. R. Hansen and Z. Chaochen. Duration calculus: Logical foundations. *Formal Asp. Comput.*, 9(3):283–330, 1997.

[18] P. Inverardi, P. Pelliccione, and M. Tivoli. Towards an assume-guarantee theory for adaptable systems. *Software Engineering for Adaptive and Self-Managing Systems, International Workshop on*, 0:106–115, 2009.

[19] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 15–24, New York, NY, USA, 1995. ACM.

[20] R. Kazhamiakin, A. Metzger, and M. Pistore. Towards correctness assurance in adaptive service-based applications. In *ServiceWave 2008*, number 5377 in LNCS. Springer, 10-13 December 2008.

[21] R. Kazhamiakin, P. Pandya, and M. Pistore. Representation, verification, and computation of timed properties. *International Conference on Web Services*, pages 497–504, 2006.

[22] R. Kazhamiakin and M. Pistore. Static verification of control and data in web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 83–90, Washington, DC, USA, 2006. IEEE Computer Society.

[23] Z. Lu, S. Li, A. Ghose, and P. Hyland. Extending semantic web service description by service assumption. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 637–643, Washington, DC, USA, 2006. IEEE Computer Society.

[24] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 257–265, Washington, DC, USA, 2005. IEEE Computer Society.

[25] S. Nakajima. Model-checking verification for reliable web service. *OOPSLA Workshop on Object-Oriented Web Services (OOWS 2002)*, 2002.

[26] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 77–88, New York, NY, USA, 2002. ACM.

[27] G. Newton, J. Pollock, and D. L. McGuinness. Semantic web rule language (SWRL), 2004.

[28] E. D. Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):257–402, 2008.

[29] M. Pistore and P. Traverso. Assumption-based composition and monitoring of web services. In *Test and Analysis of Web Services*, pages 307–335, 2007.

[30] A. van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 5–19, New York, NY, USA, 2000. ACM.

[31] A. van Lamsweerde, E. Letier, and R. Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, 1998.

Formal Analysis and Verification of Self-Healing Systems[⋆]

Hartmut Ehrig[1], Claudia Ermel[1], Olga Runge[1],
Antonio Bucchiarone[2] and Patrizio Pelliccione[3]

[1] Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{ehrig, lieske, olga}@cs.tu-berlin.de
[2] FBK-IRST, Trento, Italy
bucchiarone@fbk.eu
[3] Dipartimento di Informatica Università dell'Aquila, Italy
patrizio.pelliccione@di.univaq.it

Abstract. Self-healing (SH) systems are characterized by an automatic discovery of system failures, and techniques how to recover from these situations. In this paper, we show how to model SH systems using algebraic graph transformation. These systems are modeled as typed graph grammars enriched with graph constraints. This allows not only for formal modelling of consistency and operational properties, but also for their formal analysis and verification using the tool AGG. As main results, we present sufficient static conditions for self-healing properties, deadlock-freeness and liveness of SH-systems. The overall approach is applied to a traffic light system case study, where the corresponding properties are verified.

1 Introduction

The high degree of variability that characterizes modern systems requires to design them with runtime evolution in mind. Self-adaptive systems are systems that autonomously decide how to adapt the system at runtime to the internal reconfiguration and optimization requirements or to environment changes and threats [1]. In 2001, IBM introduced the concept of autonomic computing [2], which has as main idea self-management of software components. The initial four self-* properties of self-adaptive systems are self-configuration, self-healing (following [3] we consider self-healing and self-repair as synonymous), self-optimization, and self-protection [4]. Self-configuration envisions components installation and configuration based on some high-level policies. Self-healing deals with automatic discovery of system failures, and with techniques to recover from them. Typically the runtime behavior of the system is monitored to determine whether a change

[⋆] Some of the authors are partly supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) and the Italian PRIN d-ASAP project.

is needed. Self-optimization monitors the system status and adjusts parameters to increase performance when possible. Finally, self-protection aims to detect and mitigate external threats [5].

In [6], Bucchiarone et al. modeled and verified dynamic software architectures and self-healing systems (called self-repairing systems in [6]), by means of hypergraphs and graph grammars. Based on this work, we show in this paper how to formally model self-healing systems by using algebraic graph transformations [7] and to prove consistency and operational properties. The main idea is to model SH-systems with typed graph grammars where the system rules are divided into three different kinds, namely normal, environment, and repair rules. Normal rules define the normal and ideal behavior of the system. Environment rules model all possible predictable failures. Finally, for each failure a repair rule is defined. This formalization enables the specification, analysis and verification of consistency and operational properties of SH-systems. More precisely, we present sufficient static conditions for two alternative self-healing properties, deadlock-freeness and liveness of SH-systems. The formalization of SH-systems proposed in this paper enables the formal analysis and verification of modeled properties by using different kinds of validation features of the AGG[4] tool. AGG is a well established graph transformation environment developed at TU Berlin that allows to model and verify typed graph grammars within the same environment. The overall theory is presented by use of a running example, namely an automated traffic light system controlled by means of electromagnetic spires that are buried some centimeters underneath the asphalt of car lanes.

Summarizing, the contribution of this paper is twofold: (i) we propose a way to model and formalize self-healing systems; (ii) we provide tool supported static verification techniques for self-healing system models.

The paper is organized as follows: Section 2 presents related work and motivates the paper. Section 3 presents the setting of our running example. Section 4 introduces typed attributed graph transformation as formal basis to specify and analyze self-healing systems. In Section 5 we define consistency and operational system properties. Static conditions for their verification are given in Section 6 and are used to analyze the behaviour and healing properties of the traffic light system. We conclude the paper in Section 7 with an outlook on future work. In Appendix A we present more details of our running example and Appendix B contains the proofs of our theorems.

2 Related Work

An interesting classification of modeling dimensions for self-adaptive systems can be found in [8]. The work in [9] presents a theoretical assume-guarantee framework to efficiently define under which conditions adaptation can be performed by still preserving the desired invariant. Contrarily to our approach, they aim to deal with unplanned adaptations.

[4] AGG: http://tfs.cs.tu-berlin.de/agg.

Focusing on approaches proposed to model self-healing systems, each example presented in [9] tries to abstract from runtime systems providing a high-level model that is useful useful to analyse, refine and verify them during the development. In this way, Software Architecture (SA) [10] has been used as a high-level view of the structural organization of the system [11,12]. Since a self-healing system must be able to change at runtime, *Dynamic Software Architectures* (DSAs) are very useful to emphasize that the SA evolves during runtime [13,14,15]. Many research works are focusing on the design and analysis of DSAs, applying formal methods such as graphs [6,16,17,18,19], logics [20] and process algebras [13].

Other approaches to model self-healing systems have defined precise languages to provide structural reconfiguration aspects. Among them we mention ArchWare, Rainbow, ArchStudio and Genie [14,11,12,21] that have been proposed with the objective of architecture-based dynamic adaptations.

In the community of Service Oriented Computing (SOC) various approaches supporting self-healing have been defined: triggering repairing strategies as a consequence of a requirement violation [22], and optimizing QoS of service-based applications [23,24]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [25,26] or of the context of mobile service-based applications [27].

The different proposals listed before are bound to particular languages and models. Our proposal instead is aimed at understanding the main notions behind such proposals by abstracting away from particular languages and notations. We aim to present a uniform formal representation that is abstract enough to cover most of these features. Moreover, we provide static verification techniques for self-healing system models.

3 Running Example: An Automated Traffic Light System

Let us introduce our running example, an automated Traffic Light System (TLS). The traffic light technology is based upon electromagnetic spires buried some centimeters underneath the asphalt of car lanes. The spires register traffic data and send them to other system components. The technology helps the infraction system by making it incontestable. In fact, the TLS is connected to cameras which record videos of the violations and automatically send them to the center of operations. In addition to the normal behavior, we may have failures caused by a loss of signals between traffic light or camera and supervisor. For each of the failures there are corresponding repair actions, which can be applied after monitoring the failures during run-time. For more detail concerning the functionality of the TLS, we refer to Appendix A.

The aim of our TLS model, based on typed graphs and graph grammars, is to ensure suitable self-healing properties by applying repair actions that ensure safe system runs. It should happen independently of the unpredictable dynamism of the traffic flow. What kind of repair actions are useful and lead to consistent system states without failures? What kind of safety and liveness properties can be guaranteed? We will tackle these questions in the next sections by providing a formal modelling and analysis technique based on algebraic graph transformation and continue our running example in Examples 1 – 6 below.

4 Formal Modelling of Self-healing Systems by Algebraic Graph Transformations

In this section, we show how to model SH-systems in the formal framework of algebraic graph transformation [7]. The main concepts of this framework which are relevant for our approach are typed graphs, graph grammars, transformations and constraints. Configurations of an SH-System are modeled by typed graphs.

Definition 1 (Typed Graphs). *A* graph $G = (N, E, s, t)$ *consists of a set of* nodes N*, a set of* edges E *and functions* $s, t : E \to N$ *assigning to each edge* $e \in E$ *the* source $s(e) \in N$ *and* target $t(e) \in N$*.*

A graph morphism $f : G \to G'$ *is given by a pair of functions* $f = (f_N : N \to N', f_E : E \to E')$ *which is compatible with source and target functions.*

A type graph TG *is a distinguished graph where nodes and edges are considered as node and edge types, respectively. A* TG-typed*, or short* typed graph $\overline{G} = (G, t)$ *consists of a graph* G *and a graph morphism* $t : G \to TG$*, called* typing morphism *of* G. Morphisms $f : \overline{G} \to \overline{G'}$ of typed graphs *are graph morphisms* $f : G \to G'$ *which are compatible with the typing morphisms of* G *and* G'*, i.e.* $t' \circ f = t$*.*

For simplicity, we abbreviate $\overline{G} = (G, t)$ by G in the following. Moreover, the approach is also valid for *attributed* and *typed attributed graphs* where nodes and edges can have data type attributes, as shown in our running example (see [7] for more details).

Example 1 (Traffic Light System). The type graph TG of our traffic light system TLS is given in Fig. 1. The initial state is the configuration graph in Fig. 2 which is a TG-typed graph where the typing is indicated by corresponding names, and the attributes are attached to nodes and edges. The initial state shows two traffic lights (TL), two cameras, a supervisor, and a center of operations, but no traffic up to now.

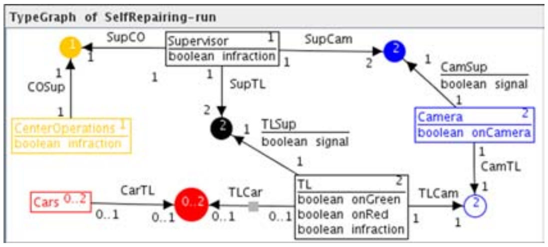


Fig. 1. TLS type graph TG

The dynamic behaviour of SH-systems is modeled by rules and transformations of a typed graph grammar in the sense of algebraic graph transformation [7].

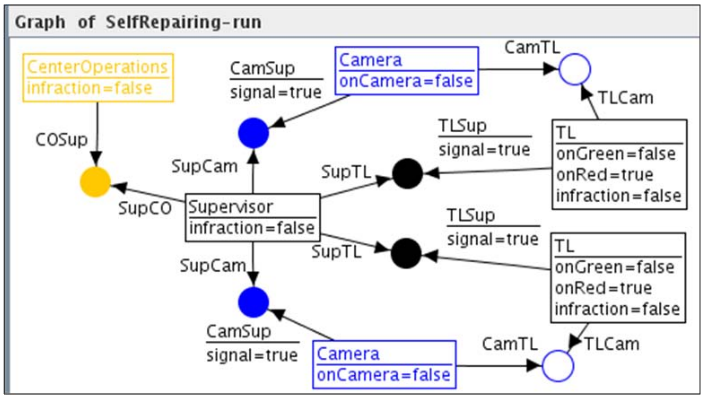


Fig. 2. TLS initial state G_{init}

Definition 2 (Typed Graph Grammar).

A typed graph grammar $GG = (TG, G_{init}, Rules)$ *consists of a type graph* TG*, a* TG*-typed graph* G_{init}*, called* initial graph*, and a set* $Rules$ *of* graph transformation rules*. Each rule* $r \in Rules$ *is given by a span* $(L \leftarrow I \rightarrow R)$*, where* L, I *and* R *are* TG*-typed graphs, called* left-hand side, right-hand side *and* interface*, respectively. Moreover,* $I \rightarrow L$*,* $I \rightarrow R$ *are injective typed graph morphisms where in most cases* I *can be considered as intersection of* L *and* R*. A rule* $r \in Rules$ *is applied to a* TG*-typed graph* G *by a* match morphism $m : L \rightarrow G$ *leading to a* direct transformation $G \overset{r,m}{\Longrightarrow} H$ *via* (r, m) *in two steps: at first, we delete the match* $m(L)$ *without* $m(I)$ *from* G *to obtain a context graph* D*, and secondly, we glue together* D *with* R *along* I *leading to a* TG*-typed graph* H*.*

More formally, the direct transformation $G \overset{r,m}{\Longrightarrow} H$ *is given by two* pushout diagrams (1) *and* (2) *in the category* $\mathbf{Graphs_{TG}}$ *of* TG*-typed graphs, where diagram* (1) *(resp.* (2)*) corresponds to gluing* G *of* L *and* D *along* I *(resp. to gluing* H *of* R *and* D *along* I*).*

$$\begin{tikzcd}
N \arrow[dr, "q"', "|" marking] & L \arrow[l, "nac"'] \arrow[d, "m"'] & I \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dl, phantom, "(1)"] & R \arrow[d, "m^*"] \arrow[dl, phantom, "(2)"] \\
 & G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Note that pushout diagram (1) *in step 1 only exists if the match* m *satisfies a* gluing condition *w.r.t. rule* r *which makes sure that the deletion in step 1 leads to a well-defined* $TG-$*typed graph* D*. Moreover, rules are allowed to have* Negative Application Conditions *(NACs) given by a typed graph morphism* $nac : L \rightarrow N$*. In this case, rule* r *can only be applied at match* $m : L \rightarrow G$ *if there is no injective morphism* $q : N \rightarrow G$ *with* $q \circ nac = m$*. This means intuitively that* r *cannot be applied to* G *if graph* N *occurs in* G*. A transformation* $G_0 \overset{*}{\Longrightarrow} G_n$ *via*

Rules in GG *consists of* $n \geq 0$ *direct transformations* $G_0 \Longrightarrow G_1 \Rightarrow ... \Rightarrow G_n$ *via rules* $r \in Rules$*. For* $n \geq 1$ *we write* $G_0 \overset{+}{\Longrightarrow} G_n$*.*

Example 2 (Rules of TLS). A rule $r = (L \leftarrow I \rightarrow R)$ of TLS with NAC $nac : L \rightarrow N$ is given in Fig. 3 where the graphs N, L and R are given from left to right (the interface I is not shown and consists of the nodes and edges which are present in both L and R, as indicated by equal numbers). For simplicity, we only show the additional part NAC of N in the left which extends L. All graph morphisms are inclusions. Rule r can be applied to graph G in Fig. 2 where the node $(1 : TL)$ in L is mapped by m to the upper node TL in G_{init}. This leads to a graph H where the attributes of this node are changed and the "car-subgraph" of R is attached to this node. Altogether, we have a direct transformation $G \overset{r,m}{\Longrightarrow} H$.

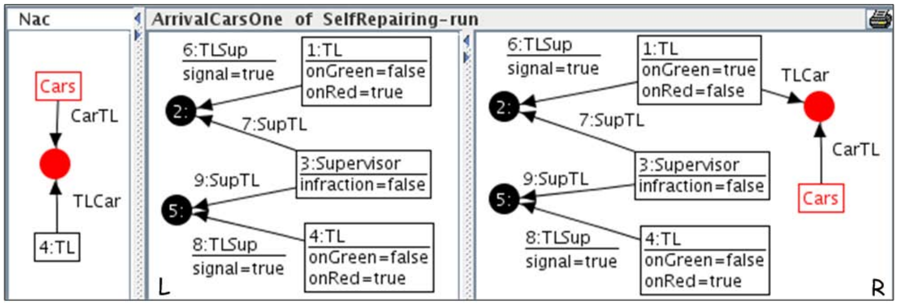


Fig. 3. TLS rule *ArrivalCarsOne*

In order to model consistency and failure constraints of a SH-system, we use graph constraints. A *TG-typed graph constraint* is given by a TG-typed graph C which is *satisfied by a TG-typed graph* G, written $G \models C$, if there is an injective graph morphism $f : C \rightarrow G$. Graph constraints can be negated or combined by logical connectors (e.g. $\neg C$). Now we are able to define SH-systems by a typed graph grammar where four kinds of rules are distinguished, called *system*, *normal*, *environment* and *repair* rules. Moreover, we have two kinds of TG-typed graph constraints, namely *consistency* and *failure* constraints.

Definition 3 (Self-healing System in GT-Framework). *A Self-healing System (SH-system) is given by* $SHS = (GG, C_{sys})$*, where:*

- $GG = (TG, G_{init}, R_{sys})$ *is a typed graph grammar with type graph* TG*, a* TG*-typed graph* G_{init}*, called* initial state*, a set of* TG*-typed rules* R_{sys} *with NACs, called* system rules*, defined by* $R_{sys} = R_{norm} \cup R_{env} \cup R_{rpr}$*, where* R_{norm} *(called* normal rules*),* R_{env} *(called* environment rules*) and* R_{rpr} *(called* repair rules*) are pairwise disjoint.*

- C_{sys} *is a set of TG-typed graph constraints, called* system constraints*, with* $C_{sys} = C_{consist} \cup C_{fail}$*, where* $C_{consist}$ *are called* consistency constraints *and* C_{fail} failure constraints*.*

For an SH-system, we distinguish the following kinds of reachable, consistent, failure and normal states, where reachable states split into normal and failure states.

Definition 4 (Classification of SH-System States).
Given an SH-system $SHS = (GG, C_{sys})$ *as defined above, we have*

1. $Reach(SHS) = \{G \mid G_{init} \overset{*}{\Longrightarrow} G \text{ via } R_{sys}\ \}$*, the* reachable states *consisting of all states reachable via system rules,*
2. $Consist(SHS) = \{G \mid G \in Reach(SHS) \wedge \forall C \in C_{consist} : G \models C\}$*, the* consistent states*, consisting of all reachable states satisfying the consistency constraints,*
3. $Fail(SHS) = \{G \mid G \in Reach(SHS) \wedge \exists C \in C_{fail} : G \models C\}$*, the* failure states*, consisting of all reachable states satisfying some failure constraint,*
4. $Norm(SHS) = \{G \mid G \in Reach(SHS) \wedge \forall C \in C_{fail} : G \not\models C_{fail}\}$*, the* normal states*, consisting of all reachable states not satisfying any failure constraint.*

Example 3 (Traffic Light System as SH-system). We define the Traffic Light SH-system $TLS = (GG, C_{sys})$ by the type graph TG in Fig. 2, the initial state G_{init} in Fig. 1, and the following sets of rules and constraints:

- $R_{norm} = \{$*ArrivalCarsOne*, *ArrivalCarsTwo*, *RemoveCarsOne*, *RemoveCarsTwo*, *InfractionOn*, *InfractionOff*$\}$,
- $R_{env} = \{$*FailureTL*, *FailureCam*$\}$,
- $R_{rpr} = \{$*RepairTL*, *RepairCam*$\}$,
- $C_{consist} = \{\neg allGreen, \neg allRed\}$,
- $C_{fail} = \{$*TLSupFailure*, *CamSupFailure*$\}$.

The normal rule *ArrivalCarsOne* is depicted in Fig. 3 and models that one or more cars arrive in one direction but no cars arrive in the other direction which is prohibited by the NAC in Fig. 3. Moreover, the traffic light switches to green in the direction of the arriving cars. Rule *ArrivalCarsTwo* (see Fig. 4) models the arrival of one or more cars at a red traffic light where no cars had been before, while there are already cars in the other direction. This rule causes also a change of the traffic light in both directions so that rules *RemoveCarsOne* resp. *RemoveCarsTwo* can be applied to model that one or more cars pass the traffic light in the green directions.

Rules *RemoveCarsOne* and *RemoveCarsTwo* are the inverse (with L and R exchanged) of the arrival rules in Fig. 3 and 4, and model the reduction of traffic at a traffic light. Rule *InfractionOn* is shown in Fig. 5 and models the situation that a car is passing the crossroad at a red light: the signal *infraction* of both the supervisor and the center of operations is set to true and the corresponding camera is starting to operate. The rule ensures that the corresponding camera is connected, using the edge attribute signal = true for edge 13:CamSup. Rule

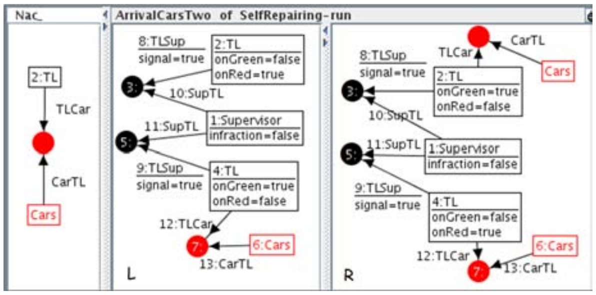


Fig. 4. TLS rule *ArrivalCarsTwo*

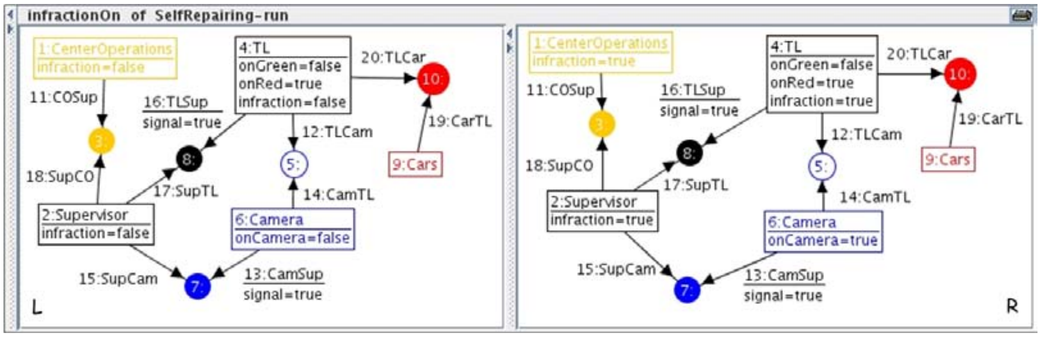


Fig. 5. Normal rule *InfractionOn* of *TLS*

InfractionOff models the inverse action (the infraction attribute is set back to false, and the camera stops running).

The environment rules are shown in Fig. 6. They model the signal disconnection of a traffic light and a camera, respectively. The repair rules, defined as inverse rules of the environment rules, model the inverse actions and set the signal attributes back to true.

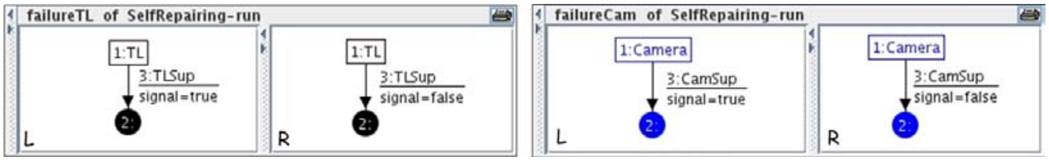


Fig. 6. Environment rules *FailureTL* and *FailureCam* of *TLS*

The consistency constraints model the desired properties that we always want to have crossroads with at least one direction showing red lights ($\neg$ *allGreen*) and avoiding all traffic lights red when there is traffic (¬ *allRed*). The corresponding constraint graphs (without negation) are shown in Fig. 7.

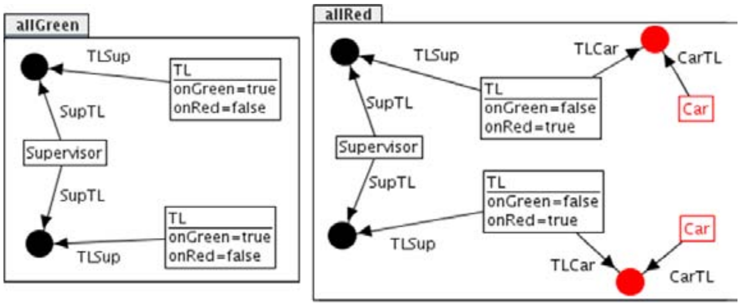


Fig. 7. Consistency constraint graphs of *TLS*

The failure constraints *TLSupFailure* and *CamSupFailure* express that either a traffic light or a camera is disconnected (the constraint graphs correspond to the right-hand sides of the environment rules in Fig. 6).

5 Consistency and Operational Properties of SH-Systems

In this section, we define desirable consistency and operational properties of SH-Systems. We distinguish system consistency, where all reachable states are consistent, and normal state consistency, where the initial state G_{init} and all states reachable by normal rules are normal states. Environment rules, however, may lead to failure states, which should be repaired by repair rules. We start with consistency properties:

Definition 5 (Consistency Properties). *An SH-System SHS is called*

1. *system consistent, if all reachable states are consistent, i.e.* $Reach(SHS) = Consist(SHS)$*;*
2. *normal state consistent, if the initial state is normal and all normal rules preserve normal states, i.e.* $G_{init} \in Norm(SHS)$ *and* $\forall G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{norm}$ $[\ G_0 \in Norm(SHS) \Rightarrow G_1 \in Norm(SHS)\ ]$

Example 4 (Consistency Properties of TLS). The SH-System TLS is system consistent, because for all $C \in C_{consist}$ $G_{init} \models C$ and for all $G_0 \overset{p}{\Longrightarrow} G_1$ via $p \in R_{sys}$ and $G_0 \in Consist(SHS)$ we also have $G_1 \in Consist(SHS)$. Similarly, TLS is normal state consistent, because $G_{init} \in Norm(SHS)$ and for all $G_0 \overset{p}{\Longrightarrow} G_1$ via $p \in R_{norm}$ for all $C \in C_{fail}$ $[\ G_0 \not\models C \Rightarrow G_1 \not\models C\ ]$. In both cases this can be concluded by inspection of the corresponding rules, constraints and reachable states. Moreover, there are also general conditions, which ensure the preservation of graph constraints by rules, but this discussion is out of scope for this paper.

Now we consider the operational properties: one of the main ideas of SH-Systems is that they are monitored in regular time intervals by checking, whether the current system state is a failure state. In this case one or more failures have occurred in the last time interval, which are caused by failure rules, provided that we have normal state consistency. With our self-healing property below we require that each failure state can be repaired leading again to a normal state. Moreover, strongly self-healing means that the normal state after repairing is the same as if no failure and repairing would have been occurred.

Definition 6 (Self-healing Properties). *An SH-System SHS is called*

1. *self-healing, if each failure state can be repaired, i.e.*
   $\forall G_{init} \Rightarrow^* G$ *via* $(R_{norm} \cup R_{env})$ *with* $G \in Fail(SHS)$
   $\exists\ G \Rightarrow^+ G'$ *via* R_{rpr} *with* $G' \in Norm(SHS)$
2. *strongly self-healing, if each failure state can be repaired strongly, i.e.*
   $\forall G_{init} \Rightarrow^* G$ *via* $(p_1 \dots p_n) \in (R_{norm} \cup R_{env})^*$ *with* $G \in Fail(SHS)$
   $\exists\ G \Rightarrow^+ G'$ *via* R_{rpr} *with* $G' \in Norm(SHS)$ *and*
   $\exists\ G_{init} \Rightarrow^* G'$ *via* $(q_1 \dots q_m) \in R^*_{norm}$,
   where $(q_1 \dots q_m)$ *is subsequence of all normal rules in* $(p_1 \dots p_n)$.

Remark 1. By definition, each strongly self-healing SHS is also self-healing, but not vice versa. The additional requirement for strongly self-healing means, that the system state G' obtained after repairing is not only normal, but can also be generated by all normal rules in the given mixed sequence $(p_1 \dots p_n)$ of normal and environment rules, as if no environment rule would have been applied. We will see that our SH-System TLS is strongly self-healing, but a modification of TLS, which counts failures, even if they are repaired later, would only be self-healing, but not strongly self-healing.

Another important property of SH-Systems is deadlock-freeness, meaning that no reachable state is a deadlock. A stronger liveness property is strong cyclicity, meaning that each pair of reachable states can be reached from each other. Note that this is stronger than cyclicity meaning that there are cycles in the reachability graph. Strong cyclicity, however, implies that each reachable state can be reached arbitrarily often. This is true for the TLS system, but may be false for other reasonable SH-Systems, which may be only deadlock-free. Moreover, we consider “normal deadlock-freeness” and “normal strong cyclicity”, where we only consider normal behavior defined by normal rules.

Definition 7 (Deadlock-Freeness and Strong Cyclicity Properties). *An SH-System SHS is called*

1. *deadlock-free, if no reachable state is a deadlock, i.e.*
   $\forall G_0 \in Reach(SHS)\ \exists\ G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{sys}$
2. *normal deadlock-free, if no state reachable via normal rules is a (normal) deadlock, i.e.* $\forall G_{init} \Rightarrow^* G_0$ *via* $R_{norm}\ \exists\ G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{norm}$
3. *strongly cyclic, if each pair of reachable states can be reached from each other, i.e.* $\forall G_0, G_1 \in Reach(SHS)\ \exists\ G_0 \Rightarrow^* G_1$ *via* R_{sys}

4. *normally cyclic, if each pair of states reachable by normal rules can be reached from each other by normal rules, i.e.* $\forall G_{init} \Rightarrow^* G_0$ *via* R_{norm} *and* $G_{init} \Rightarrow^* G_1$ *via* R_{norm} *we have* $\exists\ G_0 \Rightarrow^* G_1$ *via* R_{norm}

Remark 2. If we have at least two different reachable states (rsp. reachable by normal rules), then "strongly cyclic" (rsp. "normally cyclic") implies "deadlock-free" (rsp. "normal deadlock-free"). In general properties 1 and 2 as well as 3 and 4 are independent from each other. But in Thm. 3 we will give sufficient conditions s.t. "normal deadlock-free" implies "deadlock-free" (rsp. "normally cyclic" implies "strongly cyclic" in Thm. 4).

6 Analysis and Verification of Operational Properties

In this section, we analyze the operational properties introduced in section 5 and give static sufficient conditions for their verification. The proofs of our theorems are given in Appendix B.

First, we define direct and normal healing properties, which imply the strong self-healing property under suitable conditions in Thm. 1. In a second step we give static conditions for the direct and normal healing properties in Thm. 2, which by Thm. 1 are also sufficient conditions for our self-healing properties. Of course, we have to require that for each environment rule, which may cause a failure there are one or more repair rules leading again to a state without this failure, if they are applied immediately after its occurrence. But in general we cannot apply the repair rules directly after the failure, because other normal and environment rules may have been applied already, before the failure is monitored. For this reason we require in Thm. 1 that each pair (p, q) of environment rules p and normal rules q is sequentially independent. By the Local Church-Rosser theorem for algebraic graph transformation [7](Thm 5.12) sequential independence of (p, q) allows to switch the corresponding direct derivations in order to prove Thm. 1. For the case with nested application conditions including NACs we refer to [28]. Moreover, the AGG tool can calculate all pairs of sequential independent rules with NACs before runtime.

Definition 8 (Direct and Normal Healing Properties). *An SH-System SHS has the*

1. *direct healing property, if the effect of each environment rule can be repaired directly, i.e.* $\forall G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{env}$ $\exists\ G_1 \overset{p'}{\Longrightarrow} G_0$ *via* $p' \in R_{rpr}$
2. *normal healing property, if the effect of each environment rule can be repaired up to normal transformations, i.e.* $\forall G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{env}$ $\exists\ G_1 \Rightarrow^+ G_2$ *via* R_{rpr} *s.t.* $\exists\ G_0 \Rightarrow^* G_2$ *via* R_{norm}

Remark 3. The direct healing property allows one to repair each failure caused by an environment rule directly by reestablishing the old state G_0. This is not required for the normal healing property, but it is required only that the repaired state G_2 is related to the old state G_0 by a normal transformation. Of course, the direct healing property implies the normal one using $G_2 = G_0$.

Theorem 1 (Analysis of Self-healing Properties). *An SH-System SHS is*

I. strongly self-healing, if we have properties 1, 2, and 3 below
II. self-healing, if we have properties 1, 2 and 4 below

1. *SHS is normal state consistent*
2. *each pair $(p, q) \in R_{env} \times R_{norm}$ is sequentially independent*
3. *SHS has the direct healing property*
4. *SHS has the normal healing property*

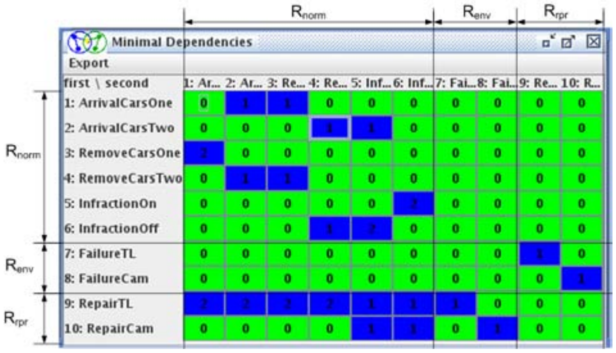
In the following Thm. 2 we give static conditions for direct and normal healing properties. In part 1 of Thm. 2 we require that for each environment rule p the inverse rule p^{-1} is isomorphic to a repair rule p'. Two rules are isomorphic if they are componentwise isomorphic. For $p = (L \leftarrow I \rightarrow R)$ with negative application condition $nac : L \rightarrow N$ it is possible (see [7] Remark 7.21) to construct $p^{-1} = (R \leftarrow I \rightarrow L)$ with equivalent $nac' : R \rightarrow N'$. In part 2 of Thm. 2 we require as weaker condition that each environment rule p has a corresponding repair rule p', which is not necessarily inverse to p. It is sufficient to require that we can construct a concurrent rule $p *_R p'$ which is isomorphic to a normal rule p''. For the construction and corresponding properties of inverse and concurrent rules, which are needed in the proof of Thm. 2 we refer to [7] Thm 5.23, [28].

Theorem 2 (Static Conditions for Direct/Normal Healing Properties).

1. *An SH-System SHS has the direct healing property, if for each environment rule there is an inverse repair rule, i.e. $\forall p \in R_{env} \; \exists \, p' \in R_{rpr}$ with $p' \cong p^{-1}$*
2. *An SH-System SHS has the normal healing property if for each environment there is a corresponding repair rule in the following sense:*
   $\forall p = (L \leftarrow K \rightarrow R) \in R_{env}$ we have
   a) *repair rule $p' = (L' \leftarrow^{l'} K' \rightarrow^{r'} R')$ with l' bijective on nodes, and*
   b) *an edge-injective morphism $e : L' \rightarrow R$ leading to concurrent rule $p *_R p'$, and*
   c) *normal rule $p'' \in R_{norm}$ with $p *_R p' \cong p''$*

Remark 4. By combining Thm. 1 and Thm. 2 we obtain static conditions ensuring that an SH-System SHS is strongly self-healing and self-healing, respectively.

Example 5 (Direct Healing Property of TLS). TLS has direct healing property because “RepairTL” rsp. “RepairCam” are inverse to “FailureTL” resp. “FailureCam” and each pair $(p, q) \in R_{env} \times R_{norm}$ is sequentially independent according to the dependency matrix of TLS in Fig. 8.



Minimal Dependencies

Export

first \ second	1: Ar...	2: Ar...	3: Re...	4: Re...	5: Inf...	6: Inf...	7: Fai...	8: Fai...	9: Re...	10: R...
1: ArrivalCarsOne	0	1	1	0	0	0	0	0	0	0
2: ArrivalCarsTwo	0	0	0	1	1	0	0	0	0	0
3: RemoveCarsOne	2	0	0	0	0	0	0	0	0	0
4: RemoveCarsTwo	0	1	1	0	0	0	0	0	0	0
5: InfractionOn	0	0	0	0	0	2	0	0	0	0
6: InfractionOff	0	0	0	1	2	0	0	0	0	0
7: FailureTL	0	0	0	0	0	0	0	0	1	0
8: FailureCam	0	0	0	0	0	0	0	0	0	1
9: RepairTL	2	2	2	2	1	1	1	0	0	0
10: RepairCam	0	0	0	0	1	1	0	1	0	0

Fig. 8. Dependency Matrix of TLS in AGG

In the following Thm. 3 and Thm. 4 we give sufficient conditions for deadlock-freeness and strong cyclicity which are important liveness properties. Here we mainly use a stepwise approach. We assume to have both properties for normal rules and give additional static conditions to conclude the property for all system rules. The additional conditions are sequentially and parallel independence and a direct correspondence between environment and repair rules, which should be inverse to each other. Similar to sequential independence, also parallel independence of rules (p, q) can be calculated by the AGG tool before runtime.

Theorem 3 (Deadlock-Freeness). *An SH-System SHS is deadlock-free, if*

1. *SHS is normally deadlock-free, and*
2. *Each pair $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ is sequentially and parallel independent.*

Theorem 4 (Strong Cyclicity). *An SH-System SHS is strongly cyclic, given*
I. properties 1 and 2, or
II. properties 1, 3 and 4 below.

1. *For each environment rule there is an inverse repair rule and vice versa.*
2. *For each normal rule there is an inverse normal rule.*
3. *SHS is normally cyclic.*
4. *Each pair $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ is sequentially independent.*

Remark 5. In part I of Thm. 4, we avoid the stepwise approach and any kind of sequential and parallel independence by the assumption that also all normal rules have inverses, which is satisfied for our TLS.

Example 6 (Strong Cyclicity and Deadlock-Freeness of TLS).

We use part I of Thm. 4 to show strong cyclicity. Property 1 is satisfied because “FailureTL” and “RepairTL” as well as “FailureCam” and “RepairCam” are inverse to each other. Property 2 is satisfied because “ArrivalCarsOne(Two)” and “RemoveCarsOne(Two)” as well as “InfractionOn” and “InfractionOff” are inverse to each other. Moreover, deadlock-freeness of TLS follows from strong cyclicity by remark 2. Note that we cannot use part II of Thm. 4 for our example TLS, because e.g. ("RepairTL", "ArrivalCarsOne") is not sequentially independent.

7 Conclusion

In this paper, we have modeled and analyzed self-healing systems using algebraic graph transformation and graph constraints. We have distinguished between consistency properties, including system consistency and normal state consistency, and operational properties, including self-healing, strongly self-healing, deadlock-freeness, and strong cyclicity. The main results concerning operational properties are summarized in Fig. 9, where most of the static conditions in Thms. 1- 4 can be automatically checked by the AGG tool.

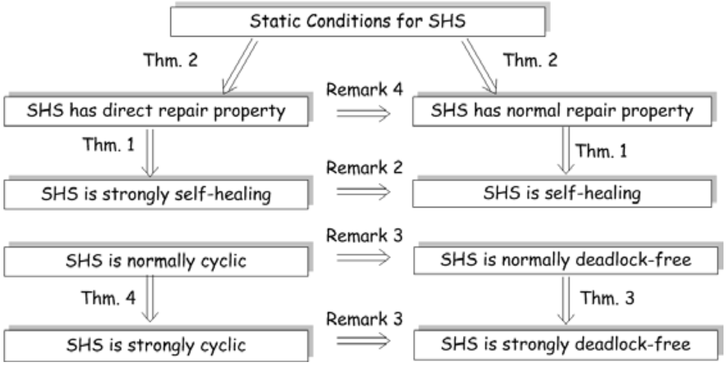

Static Conditions for SHS
Thm. 2
Thm. 2
SHS has direct repair property
Remark 4
SHS has normal repair property
Thm. 1
Thm. 1
SHS is strongly self-healing
Remark 2
SHS is self-healing
SHS is normally cyclic
Remark 3
SHS is normally deadlock-free
Thm. 4
Thm. 3
SHS is strongly cyclic
Remark 3
SHS is strongly deadlock-free

Fig. 9. Operational properties of self-healing systems

All properties are verified for our traffic light system. Note that in this paper, the consistency properties are checked by inspection of corresponding rules, while the operational properties are verified using our main results. In future work we will also provide an analysis and verification of the consistency properties using the theory of graph constraints and negative/nested application conditions in [28,29]. Moreover, we will investigate how far the techniques in this paper for self-healing systems can be used and extended for more general self-adaptive systems.

References

1. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. (2009) 48–70

2. Horn, P.: Autonomic computing: IBM's Perspective on the State of Information Technology (2001)
3. Rodosek, G.D., Geihs, K., Schmeck, H., Burkhard, S.: Self-healing systems: Foundations and challenges. In: Self-Healing and Self-Adaptive Systems. Number 09201 in Dagstuhl Seminar Proceedings, Germany (2009)
4. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1) (2003) 41–50
5. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Segal, A., Kephart, J.O.: Autonomic computing: Architectural approach and prototype. Integr. Comput.-Aided Eng. **13**(2) (2006) 173–188
6. Bucchiarone, A., Pelliccione, P., Vattani, C., Runge, O.: Self-repairing systems modeling and verification using agg. In: WICSA'09. (2009)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer (2006)
8. Andersson, J., Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Software Engineering for Self-Adaptive Systems. Springer-Verlag (2009) 27–47
9. Inverardi, P., Pelliccione, P., Tivoli, M.: Towards an assume-guarantee theory for adaptable systems. In: SEAMS, IEEE Computer Society (2009) 106–115
10. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. SIGSOFT Softw. Eng. Notes **17**(4) (1992) 40–52
11. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE. (2007) 259–268
12. Taylor, R.N., van der Hoek, A.: Software design and architecture the once and future focus of software engineering. In: FOSE. (2007) 226–243
13. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: FASE'98. (1998)
14. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: WOSS '02, ACM (2002) 27–32
15. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software **23**(2) (2006) 62–70
16. Métayer, D.L.: Describing software architecture styles using graph grammars. IEEE Trans. Software Eng. **24**(7) (1998) 521–533
17. Hirsch, D., Inverardi, P., Montanari, U.: Modeling software architecutes and styles with graph grammars and constraint solving. In: WICSA. (1999) 127–144
18. Baresi, L., Heckel, R., Thone, S., Varro, D.: Style-based refinement of dynamic software architectures. In: WICSA'04, IEEE Computer Society (2004)
19. Bucchiarone, A.: Dynamic software architectures for global computing systems. PhD thesis, IMT Institute for Advanced Studies, Lucca, Italy (2008)
20. Aguirre, N., Maibaum, T.S.E.: Hierarchical temporal specifications of dynamically reconfigurable component based systems. ENTCS **108** (2004) 69–81
21. Bencomo, N., Blair, G.S.: Using architecture models to support the generation and operation of component-based adaptive systems. In: Software Engineering for Self-Adaptive Systems. (2009) 183–200
22. Spanoudakis, G., Zisman, A., Kozlenkov, A.: A service discovery framework for service centric systems. In: IEEE SCC. (2005) 251–259
23. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: GECCO. (2005) 1069–1075
24. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: WWW. (2003) 411–421

25. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: ESSPE'07, ACM (2007) 11–20
26. Colombo, M., Nitto, E.D., Mauri, M.: Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In: ICSOC. (2006) 191–202
27. Rukzio, E., Siorpaes, S., Falke, O., Hussmann, H.: Policy based adaptive services for mobile commerce. In: WMCS'05, IEEE Computer Society (2005)
28. Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. In Drewes, F., Habel, A., Hoffmann, B., Plump, D., eds.: Manipulation of Graphs, Algebras and Pictures: Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday. (2009)
29. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, Universität Oldenburg (2009)

A Details of Running Example: An Automated Traffic Light System

The TLS is responsible for regulating the traffic lights in a "smart" way. In particular, it provides the following functionalities:
Switching the traffic light signal: The spires send signals created by the passing vehicles to an Electronic Control Unit (ECU) which registers the medium speed of the cars, hence the intensity of traffic. The ECU component sends the data to a component called Supervisor. The Supervisor imparts a cadence to the green and the red lights of several traffic lights in the same street. The system gives a priority if more cars are in line in the same direction.
Management of infractions: Each camera is connected to the Supervisor, which constantly controls the traffic light signal and the ECU. If a vehicle transits upon the spires while the street light is red, the Supervisor triggers the camera, which starts recording. When the light is green, it ignores the spires and does not trigger the cameras. Contextually with the infraction event, the camera sends the records to the video recorder of the *CenterOfOperations*, which stores date, time and Supervisor ID in order to avoid legal challenges.
Error management: In regular time intervals, the system is monitored for errors or failures which are sent to the *CenterOfOperations*. Errors can be the break down of the components as well as the loss of connection from one of the components of the TLS to the rest of the system. The *CenterOfOperations* triggers a repairing procedure to recover from system faults.

In this scenario the dynamism is given by the traffic flow. Cars join and leave the system continuously and there is no way to predict it, so we cannot predict the traffic light behavior. In the following we give the TLS requirements:

- *Traffic flow:* As the cars get to the cross road, the traffic flow is stored in the *Spire* component and sent to the *ECU* component. Then, the *ECU* component forwards the data to the *Supervisor* component which will manage it;
- *Light:* For each system configuration, each crossroad has at most one green light turned on.
- *Broken camera:* the Supervisor component checks in regular time intervals if there is an error signal linked to the *Camera* component, and the *CenterOfOperations* component takes care to repair it;
- *Broken traffic light:* if there is a loss of a traffic light signal, the system handles it in order to repair it;
- *No traffic flow:* when there is no traffic flow then every traffic light must be turned to red;
- *No traffic blocking:* when there is traffic then it must be avoided that all traffic lights are red.

B Proofs

Theorem 1 (Analysis of Self-healing Properties) *An SH-System SHS is*

I. strongly self-healing, if we have properties 1, 2, and 3 below
II. self-healing, if we have properties 1, 2 and 4 below

1. *SHS is normal state consistent*
2. *each pair $(p, q) \in R_{env} \times R_{norm}$ is sequentially independent*
3. *SHS has the direct healing property*
4. *SHS has the normal healing property*

Proof. I. Given $G_{init} \Rightarrow^* G$ via $(p_1, \dots p_n) \in (R_{norm} \cup R_{env})^*$ with $G \in Fail(SHS)$ we have $n \geq 1$, because $G_{init} \in Norm(SHS)$ by 1. By sequential independence in 2. we can switch the order of $(p_1, \dots p_n)$, s.t. first all normal rules $p_i \in R_{norm}$ and then all environment rules $p_i \in R_{env}$ are applied.
As example let us consider $G_{init} \Rightarrow^+ G$ via $(q_1, p_1, q_2, p_2, q_3)$ with $q_i \in R_{norm}$ and $p_i \in R_{env}$. Then sequential independence leads by the Local Church-Rosser theorem to equivalent sequences in subdiagram (1), (2), (3) respectively.

$$\begin{tikzcd}
G_{init} \arrow[r, Rightarrow, "q_1"] & G_1 \arrow[r, Rightarrow, "p_1"] \arrow[dr, Rightarrow, "q_2"'] & G_2 \arrow[r, Rightarrow, "q_2"] \arrow[d, phantom, "(1)"] & G_3 \arrow[r, Rightarrow, "p_2"] \arrow[dr, Rightarrow, "q_3"'] & G_4 \arrow[r, Rightarrow, "q_3"] \arrow[d, phantom, "(2)"] & G \arrow[dl, Rightarrow, bend left, "p'_2"] \\
 & & G'_2 \arrow[ur, Rightarrow, "p_1"'] \arrow[dr, Rightarrow, "q_3"'] & (3) & G'_4 \arrow[ur, Rightarrow, "p_2"'] \arrow[dl, Rightarrow, bend left, "p'_1"] & \\
 & & & G'_3 = G' \arrow[ur, Rightarrow, "p_1"'] & &
\end{tikzcd}$$

By the direct healing property of 3, we have $p'_1, p'_2 \in R_{rpr}$ with $G'_4 \overset{p'_1}{\Longrightarrow} G'_3$ and $G \overset{p'_2}{\Longrightarrow} G'_4$. With $G' = G'_3$ we have $G \Rightarrow^+ G'$ via $(p'_2, p'_1) \in R^*_{rpr}$ and $G_{init} \Rightarrow^* G'$ via $(q_1, q_2, q_3) \in R^*_{norm}$ where (q_1, q_2, q_3) is the subsequence of $(q_1, p_1, q_2, p_2, q_3)$ of all normal rules and normal state consistency of 1. implies $G_{init}, G' \in Norm(SHS)$.
Note that in the repair sequence (p'_2, p'_1) the (possible) failures caused by $p_1, p_2 \in R_{env}$ are repaired in opposite order. In general the sequence $(p_1, \dots p_n)$ $(n \geq 1)$ contains at least one rule in R_{env}, because otherwise $G \notin Fail(SHS)$ by 1. This implies that we have a repair sequence $G \Rightarrow^+ G'$ via R_{rpr} of length $n \geq 1$. Hence SHS is strongly self-healing.
II. We can proceed as above up to the point that first all normal and then all environment rules are applied. As shown in our example the normal healing property 4 leads first to a repair transformation $G \Rightarrow^+ G_5$ via R_{rpr} with $G'_4 \Rightarrow^* G_5$ via R_{norm}, then we can switch rules in (4) according to 2. Finally the normal healing property leads to $G_5 \Rightarrow^+ G_6$ via R_{rpr} with $G'_5 \Rightarrow^* G_6$ via R_{norm}.

$$\begin{tikzcd}
G_{init} \arrow[r, Rightarrow, "q_1"] & G_1 \arrow[r, Rightarrow, "p_1"] \arrow[dr, Rightarrow, "q_2"'] & G_2 \arrow[r, Rightarrow, "q_2"] \arrow[d, phantom, "(1)"] & G_3 \arrow[r, Rightarrow, "p_2"] \arrow[dr, Rightarrow, "q_3"'] & G_4 \arrow[r, Rightarrow, "q_3"] \arrow[d, phantom, "(2)"] & G \arrow[r, Rightarrow, "R_{rpr}", "+"{pos=0.9}] & G_5 \arrow[r, Rightarrow, "R_{rpr}", "+"{pos=0.95}] & G_6 = G' \\
& & G'_2 \arrow[ur, Rightarrow, "p_1"'] \arrow[dr, Rightarrow, "q_3"'] \arrow[rr, phantom, "(3)"] & & G'_4 \arrow[ur, Rightarrow, "p_2"'] \arrow[urr, Rightarrow, bend right, "R_{norm}"', "*"{pos=0.9}] \arrow[drr, phantom, "(4)"] & & & \\
& & & G'_3 \arrow[ur, Rightarrow, "p_1"'] \arrow[rrr, Rightarrow, "R_{norm}"', "*"{pos=0.95}] & & & G'_5 \arrow[uu, Rightarrow, "p_1"'] \arrow[uur, Rightarrow, bend right, "R_{norm}"', "*"{pos=0.9}] &
\end{tikzcd}$$

Altogether, we obtain for $G' = G_6$ a repair transformation $G \Rightarrow^{+} G_5 \Rightarrow^{+} G_6 = G'$ and a normal rule transformation $G_{init} \Rightarrow^{*} G'_3 \Rightarrow^{*} G'_5 \Rightarrow^{*} G_6 = G'$, which implies $G' \in Norm(SHS)$ by 1. In general $G \in Fail(SHS)$ implies that we have at least one environment rule in the given sequence and hence a repair transformation $G \Rightarrow^{+} G'$ of length $n \geq 1$. Hence SHS is self-healing. □

Theorem 2 (Static Conditions for Direct / Normal Healing Properties)

1. *An SH-System SHS has the direct healing property, if for each environment rule there is an inverse repair rule, i.e.* $\forall p \in R_{env} \; \exists \; p' \in R_{rpr}$ *with* $p' \cong p^{-1}$
2. *An SH-System SHS has the normal healing property if for each environment there is a corresponding repair rule in the following sense:*
   $\forall p = (L \leftarrow K \rightarrow R) \in R_{env}$ *we have*
   a) *repair rule* $p' = (L' \leftarrow^{l'} K' \rightarrow^{r'} R')$ *with* l' *bijective on nodes, and*
   b) *an edge-injective morphism* $e : L' \rightarrow R$ *leading to concurrent rule* $p *_R p'$*, and*
   c) *normal rule* $p'' \in R_{norm}$ *with* $p *_R p' \cong p''$

Proof. 1. Given $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ with NACs $nac_{i,L} : L \rightarrow N_{i,L} (i \in I)$ the inverse rule is given by $p^{-1} = (R \xleftarrow{r} K \xrightarrow{l} L)$ with corresponding NACs $nac_{i,R} : R \rightarrow N_{i,R}$. Now given $p \in R_{env}$ with $G_0 \overset{p}{\Longrightarrow} G_1$, we have by assumption $p' \in R_{rpr}$ with $p' \cong p^{-1}$ and by construction of p^{-1} also $G_1 \overset{p^{-1}}{\Longrightarrow} G_0$ and hence also $G_1 \overset{p'}{\Longrightarrow} G_0$.
2. The concurrent rule $p^* = p *_R p'$ is given by

$$\begin{tikzcd}
L \arrow[d, "id"'] \arrow[r, phantom, "(5)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "id"] & R \arrow[dr, "id"] \arrow[d, phantom, "(6)"] & & L' \arrow[dl, "e"'] & K' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] \arrow[l, phantom, "(7)", shift right=8] \arrow[r, phantom, "(8)", shift right=8] & R' \arrow[d] \\
L^* = L & K \arrow[l] \arrow[rr] & & R & & K'' \arrow[ll] \arrow[r] & R^* \\
& & & K^* \arrow[ull] \arrow[urr] \arrow[u, phantom, "(9)"] & & &
\end{tikzcd}$$

where (5), (6) are trivial Pus Outs, Push Out-complement K'' exists in (7) because e is edge-injective and l' bijective on nodes (see prop. a)) s.t. the gluing condition is satisfied. Moreover (8) is constructed as PO and (9) as pullback (PB).

Given $G_0 \overset{p,m}{\Longrightarrow} G_1$ with m injective (see Remark 6) and $p \in R_{env}$ we have

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[r, phantom, "(1)", shift right=8] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[dr, "n"] \arrow[d, phantom, "(2)"] & & L' \arrow[ll, "e"'] \arrow[dl, "m'"'] & K' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] \arrow[l, phantom, "(3)", shift right=8] \arrow[r, phantom, "(4)", shift right=8] & R' \arrow[d] \\
G_0 & D_0 \arrow[l] \arrow[rr] & & G_1 & & D_1 \arrow[ll] \arrow[r] & G_2
\end{tikzcd}$$

where POs (1), (2) are given, leading to injective comatch n, m' defined by $m' = n \circ e$ is edge-injective by property b), PO-complement D_1 in (3) exists, because the gluing condition is satisfied, and (4) is constructed as PO.

Hence we have an R-related transformation sequence $G_0 \overset{p,m}{\Longrightarrow} G_1 \overset{p',m'}{\Longrightarrow} G_2$. Using the Concurrency Theorem (see [7] Thm 5.23) we have a corresponding transformation $G_0 \overset{p^*,m}{\Longrightarrow} G_2$ via the concurrent rule p^*. According to property c) we have $p^{''} \in R_{norm}$ with $p^* \cong p^{''}$ and hence also $G_0 \overset{p^{''}}{\Longrightarrow} G_2$ via R_{norm}. Finally, $G_1 \overset{p'}{\Longrightarrow} G_2$ via $p' \in R_{rpr}$ is the required transformation $G_1 \Rightarrow^+ G_2$ via R_{rpr}.

The sequence $G_0 \overset{p,m}{\Longrightarrow} G_1 \overset{p',m'}{\Longrightarrow} G_2$ is R-related because we have the following diagram where PO (3) splits into

$$\begin{tikzcd}
L \arrow[d, "id"'] \arrow[dr, phantom, "(5)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "id"] \arrow[drr, phantom, "(6)"] & R \arrow[dr, "id"] & & L' \arrow[dl, "e"'] \arrow[d, phantom, "(7)"] & K' \arrow[l, "l'"'] \arrow[r, "r'"] \arrow[d] \arrow[dd, dotted, bend left] \arrow[dr, phantom, "(8)"] & R' \arrow[d] \arrow[dd, dotted, bend left] \\
L^* = L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[rr] \arrow[d] \arrow[drr, phantom, "(2)"] & & R \arrow[d, "n"] \arrow[dr, phantom, "(9)"] & & K'' \arrow[ll] \arrow[r] \arrow[d, dotted] \arrow[dr, phantom, "(10)"] & R^* \arrow[d, dotted] \\
G_0 & D_0 \arrow[l] \arrow[rr] & & G_1 & & D_1 \arrow[ll] \arrow[r, dotted] & G_2
\end{tikzcd}$$

POs (7) and (9) using the PO-PB decomposition property (see [7]) with n injective. Finally (4) splits into POs (8) and (10). □

Remark 6. In part 2 of the theorem we require that $p \in R_{env}$ is applied with injective match m. If all rules in R_{sys} are required to be applied with injective matches, then we have to require that e in *Condition 2 b)* is injective. If p and p' have NACs we have to require that $m \models NAC(p)$ implies $m' \models NAC(p')$, where $m' = n \circ e$ and n is the comatch in $G_0 \overset{p,m}{\Longrightarrow} G_1$ as shown in the proof.

Theorem 3 (Deadlock-Freeness) *An SH-System SHS is deadlock-free, if*

1. *SHS is normally deadlock-free, and*
2. *Each pair $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ is sequential and parallel independent.*

Proof. Given $G_{init} \Rightarrow^* G_0$ via R_{sys} we have to show the existence of $G_0 \overset{p}{\Longrightarrow} G_1$ via some $p \in R_{sys}$. Similar to proof of Thm. 1, sequential independence allows one to construct an equivalent transformation sequence $G_{init} \Rightarrow^* G_0' \Rightarrow^* G_1$ via R_{norm} in the first and via $(R_{env} \cup R_{rpr})$ in the second part. Now normal deadlock-freeness

$$\begin{tikzcd}
G_{init} \arrow[rr, Rightarrow, "R_{sys}\quad *"] \arrow[dr, Rightarrow, "R_{norm}"', "*"] & & G_0 \arrow[rr, Rightarrow, "p"] & & G_1 \\
& G_0' \arrow[ur, Rightarrow, "*", "(R_{env} \cup R_{rpr})"'] \arrow[rr, Rightarrow, "p"'] & & G_1' \arrow[ur, Rightarrow, "*", "(R_{env} \cup R_{rpr})"'] &
\end{tikzcd}$$

implies the existence of $G'_0 \overset{p}{\Longrightarrow} G'_1$ via some $p \in R_{norm}$. Now parallel independence allows one to construct by the local Church-Rosser theorem $G_0 \overset{p}{\Longrightarrow} G_1$ and $G'_1 \Rightarrow^* G_0$ via $(R_{env} \cup R_{rpr})$ for some G_1. $G_0 \overset{p}{\Longrightarrow} G_1$ with $p \in R_{norm} \leq R_{sys}$ is the required transformation. □

Theorem 4 (Strong Cyclicity) *An SH-System SHS is strongly cyclic, if we have I. properties 1 and 2, or II. properties 1, 3 and 4 below.*

1. *For each environment rule there is an inverse repair rule and vice versa.*
2. *For each normal rule there is an inverse normal rule.*
3. *SHS is normally cyclic.*
4. *Each pair $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ is sequentially independent.*

Proof. I. Given conditions 1 and 2 it is sufficient to show that for each $G_{init} \Rightarrow^* G_n$ via R_{sys} there is also a reverse sequence $G_n \Rightarrow^* G_{init}$ via R_{sys}. In fact, for $G_{init} \Rightarrow^* G_n$ via $(p_1, \ldots, p_n) \in R^*_{sys}$ we have $G_n \Rightarrow^* G_{init}$ via $(p_n^{-1}, \ldots, p_1^{-1}) \in R^*_{sys}$.

II. Given conditions 1, 3 and 4, and $G_{init} \Rightarrow^* G_1$ via R_{sys} and $G_{init} \Rightarrow^* G_2$ via R_{sys}. By sequential independence (cond. 4) we can split the sequences as shown in (1) and (2). For each direct transformation $G_3 \overset{p}{\Longrightarrow} G_4$ in $G'_1 \Rightarrow^* G_1$ via $(R_{env} \cup R_{rpr})$ we have $p' \in (R_{rpr} \cup R_{env})$ with $p \cong p'^{-1}$ leading to a direct transformation $G_4 \overset{p'}{\Longrightarrow} G_3$ with $p' \in (R_{rpr} \cup R_{env})$. Hence we obtain a transformation sequence $G_1 \Rightarrow^* G'_1$ via $(R_{rpr} \cup R_{env})$. Since SHS is normally cyclic (cond. 3), we obtain $G'_1 \Rightarrow^* G'_2$ via R_{norm} and altogether $(G_1 \Rightarrow^* G_2) = (G_1 \Rightarrow^* G'_1 \Rightarrow^* G'_2 \Rightarrow^* G_2)$ via R_{sys}.

$$\begin{tikzcd}
 & & G_{init} \arrow[dll, Rightarrow, "R_{sys}"', "(1)", "*" near end] \arrow[dl, Rightarrow, "R_{norm}", "*"' near end] \arrow[dr, Rightarrow, "R_{norm}"', "*" near end] \arrow[drr, Rightarrow, "R_{sys}", "(2)"', "*"' near end] & & \\
G_1 \arrow[r, Rightarrow, bend right, "(R_{env} \cup R_{rpr})"', "*"' near end] & G'_1 \arrow[l, Rightarrow, "(R_{env} \cup R_{rpr})", "*"' near end] \arrow[rr, Rightarrow, "R_{norm}"', "*" near end] & & G'_2 \arrow[r, Rightarrow, "(R_{env} \cup R_{rpr})"', "*" near end] & G_2
\end{tikzcd}$$

This shows that SHS is strongly cyclic. □

Towards Data-Aware QoS-Driven Adaptation for Service Orchestrations

March 2010

facultad de informática

universidad politécnica de madrid

Dragan Ivanović
Manuel Carro
Manuel Hermenegildo
Pedro López-Garcia
Edison Mera

TR Number CLIP 5/2009.1

Technical Report Number: CLIP 5/2009.1
March,2010

Authors

Dragan Ivanović
idragan@clip.dia.fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)

Manuel Carro
mcarro@fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)

Manuel Hermenegildo
herme@fi.upm.es
Computer Science School
Universidad Politécnica de Madrid (UPM)
and IMDEA Software, Spain

Pedro López
pedro.lopez@imdea.org
Computer Science School
Universidad Politécnica de Madrid (UPM)
and IMDEA Software, Spain

Edison Mera
edison@fdi.ucm.es
Facultad de Informática
Universidad Complutense de Madrid (UCM)

Acknowledgements

This research has been funded by the EU 7th. FP NoE S-Cube 215483, and partially by FET IST-231620 *HATS*, EUREKA 06042 *ES_PASS*, MICINN TIN-2008-05624 *DOVES*, MIN FIT-340005-2007-14, and CM project P2009/TIC/1465 (*PROMETIDOS*).

Abstract

Several activities in service oriented computing can benefit from knowing properties of a given service composition ahead of time. We will focus here on properties related to *computational cost* and *resource usage*, in a wide sense, as they can be linked to QoS characteristics. In order to attain more accuracy, we formulate computational cost / resource usage as *functions on input data* (or appropriate abstractions thereof) and show how these functions can be used to make more informed decisions when performing composition, proactive adaptation, and predictive monitoring. We present an approach to, on one hand, automatically synthesize these functions from orchestrations and, on the other hand, to effectively use them to increase the quality of non-trivial service-based systems with data-dependent behavior. We validate our approach by means of simulations with runtime selection of services and adaptation due to service failure.

Keywords: Service Orchestrations, Resource Usage Analysis, Data Awareness, Monitoring, Adaptation.

Contents

1 Introduction

Service Oriented Computing (SOC) is a well-established paradigm which aims at expressing and exploiting the computation possibilities of loosely coupled systems which interact remotely. Such systems expose themselves via service interfaces whose description may include operation signatures, descriptions of behavior, and others, while the implementation is completely hidden. Services can be combined to accomplish more complex tasks through *service compositions*, which are usually expressed using either a general-purpose programming language or languages designed to express business processes and compositions [4, 8]. These compositions can in turn expose themselves as full-fledged services.

One distinguishing feature of SOC systems is that they are expected to be active during long periods of time and span across geographical and administrative boundaries. These characteristics require having monitoring and adaptation capabilities at the heart of SOC. Monitoring compares the actual and expected system behavior. If a too large deviation is detected, an adaptation process (which may involve, e.g., rebinding to another provider of a service) may be triggered. When deviations can be predicted before they actually happen, both monitoring and adaptation can act ahead of time (being termed, respectively, *predictive* and *proactive*), performing prevention instead of healing.

Detecting deviations requires a behavioral model, which is used to check the current behavior or to predict a future behavior. Naturally, the more precise a model is, the better adaptation / monitoring results will be achieved. In this paper we will develop and evaluate models which, based on a combination of static analysis and actual run-time data, increase accuracy by providing upper and lower approximations of computational cost / resource usage measures which can be related to QoS characteristics. For example, the number of service invocations can be related to execution time when information about network speed is available.

2 Computation Cost Analysis and Services

Computational cost analysis aims at statically determining the computational cost (in terms of, e.g., execution steps or number of instructions) of a given algorithm for some input data. Tools to perform this kind of analysis have been developed in the field of programming languages.

However, to the best of the authors' knowledge, no similar work exists for SOC, although several approaches to automatically deriving QoS characteristics for compositions have been proposed [3, 2]. While these have much in common with our proposal, they do not treat data operations or relate QoS estimation with the characteristics of input data. Instead, some execution characteristics (e.g., number of iterations in a loop) are often either fixed or modeled statistically. Also, aggregating QoS characteristics of service compositions exposed as services is often not done. Some proposals [1] aim at performing global optimization, but still ignore data-related issues. Our proposal addresses both dimensions (global information and data-sensitivity) while still aiming at a completely automatic analysis.

2.1 A Motivating Example

We illustrate the relevance of taking actual data into account when generating QoS expressions for service compositions with a motivating example.

Fig. 1 shows a fragment of a (stylized) car part reservation system. A part Provider serves its

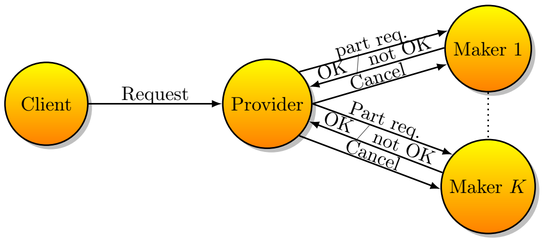


Figure 1: Simplified car part reservation system.

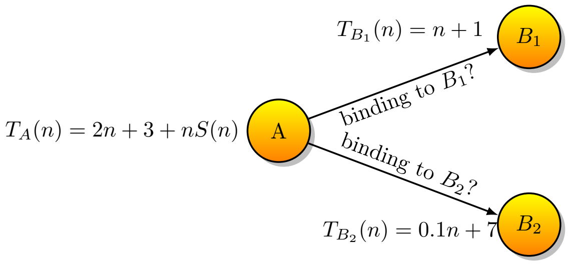


Figure 2: Invoking other services.

Client by reserving a number of part types from a pool of part Makers. The protocol only allows the Provider to reserve one part type per service invocation to a Maker. An invoked Maker replies `ok` if the part type is available and `not ok` otherwise; in this case the Provider goes to another Maker. If no Maker can reserve some car part type, the Provider cancels all previously reserved part types with a `cancel` message. Since every service invocation takes some time to complete, the number of car part types impacts the total time that Provider needs to complete a reservation for Client. Thus, a precise model of the time needed by Provider should take into account the *Request*, and more accurate time estimations should be expressed as functions on properties (e.g., number of types) of the incoming *Request* message.

2.2 Computational Cost of Service Networks

The function which results from the analysis of computational cost depends on the internal logic of the service composition (the Provider, in our example), but also on the behavior of the invoked services (the Makers), as they may, in turn, send additional messages which add to the global count.

Fig. 2 depicts this scenario in some detail. The input message is abstracted in this example as a parameter n (i.e., the number of car part types in our example) on which some measure of computational cost depends. The cost of service A is $T_A(n)$. As A invokes n times another service, (represented by a generic S), for which B_1 and B_2 are two candidates with different computational cost, its overall computational cost depends as well on which service is selected to perform the composition. Using the $T(n)$ values from Fig. 2, the computational cost corresponding to these two options would be:

$$\begin{array}{lll} T_{A_1}(n) = 2n + 3 + n(n+1) & = n^2 + 3n + 3 & \{\mathrm{AB}_1\} \\ T_{A_2}(n) = 2n + 3 + n(.1n + 7) & = 0.1n^2 + 9n + 3 & \{\mathrm{AB}_2\} \end{array}$$

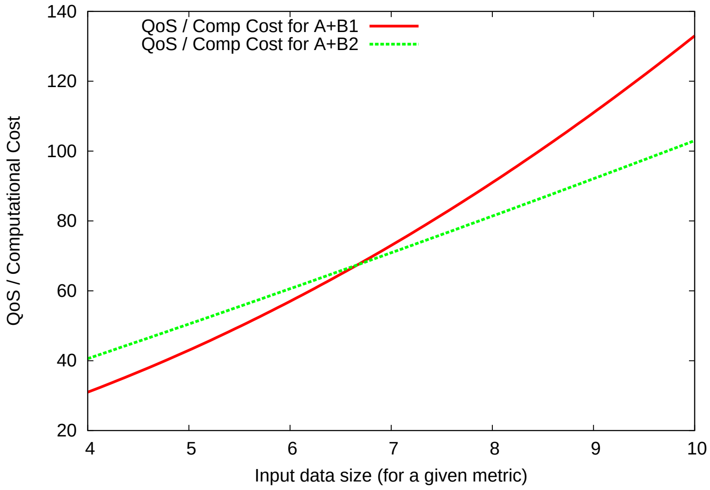


Figure 3: Computational cost, services AB_1 and AB_2.

and to decide between B_1 or B_2, T_{A_1} and T_{A_2} have to be compared (Fig. 3). This opens up the possibility of taking into account the size n of the data to select a configuration depending on the expected usage, and it requires information about B_1 and B_2 in order to automatically work out the resulting overall computational cost.

The computational cost-related information for B_1 and B_2 can be made available in much the same way as other service-related information (e.g., interfaces or XML schemes) is published. It needs to include, at least, the expected computational cost (preferably as a function of input data characteristics) and (possibly) the relationship between the sizes of the input and output data for every operation in the interface. The availability of these descriptions can make it possible to *automatically* work out T_{A_1} and T_{A_2} to compare them. In turn, A should publish the information it synthesizes, so that it can then be used by other compositions. In our view, this repeated process of synthesis, comparison, and publishing, is a step towards simultaneously achieving true dynamicity and optimal selection in the creation and adaptation of service networks.

Note that these abstract descriptions do not compromise the privacy of the implementation of the service being described, as they act as a high-level contract on the behavior of the service. Besides, in an open ecosystem of services, those which publish such descriptions would have a competitive advantage, as they make it possible for customers to make better decisions on which services to bind to.

Given a service A, if we assume that any services it invokes have a constant computational cost $T_{B_i}(n) = 1$, then the computational cost obtained for A measures how much its structure alone contributes to the total computational cost. We have termed this the *structural computational cost* of a service, and it will be used later as an approximation of the real computational cost.

Two key questions are: to which point functions expressing the cost of the computations are applicable to determining QoS, and to which point these functions can be automatically (and effectively) inferred for service compositions.

2.3 Approximating Actual Behavior

The computational cost measures we will deal with count relevant *events* which are deterministically related to the input data: processing steps, number of service invocations, size of the messages, etc. To infer such computational costs we follow the approach to resource analysis of [7] which, given data on how much a few selected basic operations contribute to the usage of some resource, tracks how many times such basic operations are performed through loops and computes the overall consumption of the resource for a complete computation. Since the number of loop iterations typically depends on the input, the overall consumption is given as a function that, for each input data size, returns (possibly upper and lower bounds to) the overall usage made of such resource for a complete computation.

Different higher-level QoS characteristics can then be derived from these functions: execution time can be approximated by aggregating the number of basic activities executed and the number of invocations, and multiplying them by an estimation of the time every (type of) activity and invocation takes; availability of a composed service can be expressed as the product of the availability of the services it invokes (assuming independence between them) and, therefore, the availability of the composition will depend on which services are invoked and how many times they are invoked, which in turn depends on the input data.

Estimations of the time used, availability, etc. of basic components are approximate and they thus introduce some noise which also makes the derived QoS functions approximations. However, because they are functions on input data they are likely to predict more accurately the behavior for a given input than a global statistical measure (we return to this later). Besides, for cases where comparison between two different QoS functions (and not their absolute value) is relevant, as in Fig. 2, the noise introduced can be expected to mutually cancel to some extent.

2.4 Upper and Lower Bounds

Automatically inferred computational cost functions can sometimes be exact, but in general only safe upper and lower bounds can be generated. These are guaranteed to be smaller than or equal to (resp. greater or equal) the function they approximate. This can be traced back to limitations of the static analysis, to the actual function depending on more parameters than, e.g., data size, and others. When these bound functions are combined with estimations to determine QoS from computational cost functions, data-aware approximations of the actual bounds are created.

While this may seem to be a disadvantage when it comes to predicting future behavior, upper / lower bounds of the actual computational cost are actually useful to *ensure* that some QoS characteristic is met, because it falls above / below the predicted threshold. As an example, Fig. 4 portrays upper and lower bound computational cost functions for two compositions for some QoS characteristic which depends on input data. Depending on the QoS meaning, we may want to make sure that we stay above or below some value. The former case needs to consider the lower bound and, conversely, the latter requires considering the upper bound. Note also that, in the example portrayed in the figure, which service will give better results clearly depends on the actual data size at run-time.

Comparing data-aware approximating functions with the probabilistic approximations used in many approaches to QoS-driven service compositions can be illustrative. Average approximations which summarize QoS characteristics in a single point clearly cannot provide behavior guarantees, as they do not provide ranges for maximum and minimum values, and they do not

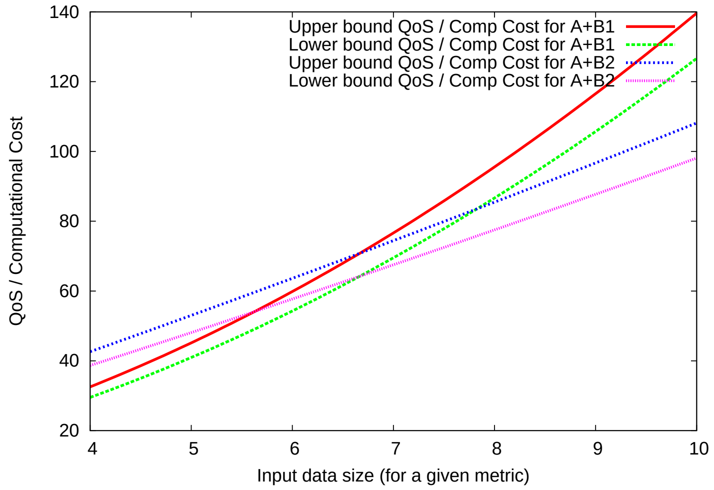


Figure 4: Using upper/lower bounds.

take data ranges into account. The statistical approach can be extended in two directions: an interval can be used to represent the maximum and minimum of the QoS, measured across all the possible input data range. But it is a coarse approximation, as it does not take into account any correlations of the QoS with input data. The other direction corresponds to using a function which, for every possible input data, represents some average value of the characteristic. This can be more precise than using a single point, but again it does not provide any bounds (not even approximate) for the QoS values.

Combining these two extensions boils down to using functions over input data which represent upper and lower bounds, and which are transformed into QoS functions by appropriately plugging in actual execution characteristics, as suggested in Section 2.3. While the results are not strictly safe, we claim that these QoS bounds can be used to predict whether the future history will stay within some predefined limits with better accuracy than just a static point, static bounds, or an average. In any of the latter cases, less information than with the upper / lower bound approximate functions is provided, so any decision will be less informed.

3 Analysis of Orchestrations

Our approach is based on translating process definitions into a language for which automatic computational cost analysis tools are available. We will now give details on this process, sketched in Fig. 5.

3.1 Overview of the Translation

Our input languages are a subset of BPEL 2.0 for the process definitions and WSDL for the associated meta-information. These are translated into an intermediate language (Table 1)

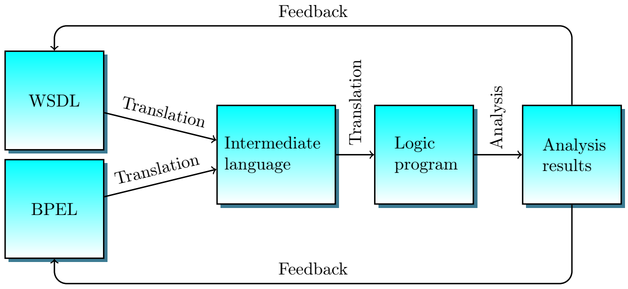


Figure 5: The overall process.

Declarations and definitions	
Complex type definition	`:-struct`(*QName*, *Members*).
Port type definition	`:-port`(*QName*,*Operations*).
External service	`:-service`(*PortName*,*Operation*, {*TrustedProperties*}).
Service definition	`service`(*Port*, *Operation*, *InMsg*, *OutMsg*)`:-`*Activity*.
Activities	
Variable assignment	*Var* `<-` *Expr*
Service invocation	`invoke`(*PortName*, *Operation*, *OutMsg*, *InMsg*).
Reply and exit	`reply`(*OutMsg*)
Sequence	$Activity_1$ **,** $Activity_2$
Conditional execution	`if`(*Cond*, *ActThen*, *ActElse*)
While loop	`while`(*Cond*, *Activity*)
Repeat-until loop	`repeatUntil`(*Activity*, *Cond*)
For-each loop	`forEach`(*Var*, *Start*, *End*, *Activity*)
Scope	`scope`(*VarDecl*, *ActivityList*)
Scope fault handler	`handler`(*FaultName*, *Activity*)
Parallel flow	`flow`(*LinkDecl*, *Activities*)
Activity in a flow	`float`(*Attributes*, *Activity*)

Table 1: Abstract orchestration elements.

which can also be used to cover other orchestration languages.[1] This intermediate representation is then translated into the *Ciao* logic programming language [6], which includes assertions to express types and input / output modes for arguments, as well as resource definitions and functions describing resource usage bounds. The resulting logic program is then analyzed by the CiaoPP tool [5], which is able to infer upper and lower bounds for computational costs [7], among other analyses.

A BPEL process definition is translated into a service definition which associates a port name and an operation with an activity that represents the orchestration body. BPEL processes forming a service network are translated into predicates which call each other to mimic service invocations.

The intermediate language can describe namespace prefixes, XML schema-derived data types

[1] Although it currently models mainly BPEL constructs.

```
:- regtype 'factory->resData'/1.
'factory->resData'('factory->resData'(A, B, C)):-
     num(A), num(B), list(C, 'factory->partInfo').

:- regtype 'factory->partInfo'/1.
'factory->partInfo'('factory->partInfo'(A, B)):-
    atm(A), atm(B).
```

Figure 6: Translation of types.

for messages, service port types, and also known properties of external services of interest to the analysis (when such services are not analyzed). The activities supported by the intermediate language include generic constructs (assignment, sequences, loops. . .) and specific constructs to model orchestration workflows: `flow`, `float`, `scope`/`handler`, and `invoke`. `flow` corresponds to the similarly named BPEL activity, while the `float` construct annotates an activity within a `flow` with a description of outgoing links and their values, join conditions based on incoming links, and a specification of the behavior in case of a join failure.

A relevant observation regarding the translation is that it does not need to follow strictly the operational semantics of the orchestration language: it has to capture enough of it to ensure that the analyzers will infer correct information while minimizing precision loss due to the translation. Despite this, in our case the translated program is executable, and mirrors quite closely (but not exactly) the operational semantics of the BPEL process under analysis.

3.2 Restrictions on Input Orchestrations

Our analysis is restricted to orchestrations which follow a *receive–reply* pattern, where all activities start after receiving an initial message and finish by dispatching either a reply or a fault notification. Additionally, we currently do not support the analysis of stateful service callbacks using correlation sets or WS-Addressing schemes. In the future we plan to relax both restrictions by identifying orchestration fragments that correspond to the *receive–reply* pattern.

In our intermediate language, we support a variant of the `scope` construct, which introduces local variables and fault / compensation handlers. We do not fully support compensation handlers, which in BPEL "undo" the effects of a successfully completed scope using snapshots of variables recorded at successful completion of the scope. Except for recording snapshots, compensation handlers can be treated as pseudo-subroutines on a scope level, and inlined at their invocation place.

3.3 Type Translation and Data Handling

The simple types in XML schemata are abstracted as three disjoint types: `numbers`, strings (translated into `atoms`), and `booleans`. Complex XML types are translated into predicates specifying how the type is built. Fig. 6 shows the translation corresponding to a fragment of the reservation scenario in Section 2.1. The type named `'factory->resData'` is a structure with three fields: two numbers and a list of elements of type `'factory->partInfo'`. Each of these elements is in turn a structure with two fields (atoms).

The accepted expression language is a subset of XPath which allows node navigation only along the descendant and attribute axes. This ensures that navigation is statically decidable and XML structures can be deforested to pass the addressed components as sepa-

A	Translation of $T([A\mid R],\eta,V)$	
`empty`	$T(R,\eta,V)$	(*Empty action*)
$A_j,\ A_k$	$T([A_j,A_k\mid R],\eta,V)$	(*Sequence*)
$\texttt{reply}(v)$	$V=\texttt{reply}(\eta(v))$	(*End of orchestration*)
$\texttt{throw}(f)$	$V=\texttt{fault}(f)$	(*No fault handler*)
	$T([H],\eta,V)$	(*Insert fault handler*)

Table 2: Inline translations.

A	Translation of $T([A\mid R],\eta,Y)$
v `<-` e	$a(\eta,Y)\leftarrow E(e,\eta,X),\ T(R,\eta\,[X/v],Y)$
$\texttt{invoke}(p,o,v,w)$	$a(\eta,Y)\leftarrow s_{p:o}(\eta(v),Z),$ $(Z=\texttt{fault}(F)\rightarrow T([\texttt{throw}(F)],\eta,Y)$ $;\ Z=\texttt{result}(X)\rightarrow T(R,\eta\,[X/w],Y))$
$\texttt{if}(c,A',A'')$	$a(\eta,Y)\leftarrow C(c,\eta),\ !,\ T([A'\mid R],\eta,Y)$ $a(\eta,Y)\leftarrow T([A''\mid R],\eta,Y)$
$\texttt{while}(c,A')$	$a(\eta,Y)\leftarrow C(c,\eta),\ !,\ T([A',A],\eta,Y)$ $a(\eta,Y)\leftarrow T(R,\eta,Y)$
$\texttt{scope}(D,A'_H)$	$a(\eta,Y)\leftarrow T([A'_H],\eta[D],Z),$ $(\texttt{var}(Z)\rightarrow T(R,\eta,Y)$ $;\ Z=\texttt{fault}(F)\rightarrow T([\texttt{throw}(F)],\eta,Y)$ $;\ Y=Z)$

Table 3: Translation into predicates.

rate arguments when necessary to improve analyzer accuracy. For example, the expression `'$req.body/item[1]/@qty'` in the intermediate language refers to the attribute `qty` of the first `item` element in the `body` part of a message stored in variable `req`. A set of standard XPath operators and basic functions, such as `position()` and `last()`, are supported.

3.4 Basic Service and Activity Translation

An orchestration that implements operation o on port p is translated into a Horn clause

$$s_{p:o}(X,Y)\leftarrow T([A],\eta,Y).$$

where X and Y correspond to the initial message and the final reply and T corresponds to the translation of a list of activities (in this case just A, the body of the orchestration). η is an environment that maps orchestration variables to logical variables, which initially just maps the input message to X. New orchestration variables may be introduced with the `scope` construct. On exit, Y can be bound to either $\texttt{reply}(R)$, where R is the contents of the reply message, or $\texttt{fault}(F)$, where F is a fault identifier.

The translation operator T accepts a list of activities and produces a Prolog goal.[2] Then $T([\,],\eta,V)=\texttt{true}$ (nothing left to translate); otherwise the case is $T([A|R],\eta,V)$ and is driven by the structure of A (Table 2). The `empty` activity is skipped. A sequence of activities is unfolded and translated one by one. A $\texttt{reply(}v\texttt{)}$ unifies the result V with the value of the reply v in the current environment. If `throw` is executed in the scope of a fault handler `H`, it is executed; otherwise the result is unified with the fault identifier.

In more complex cases (Table 3), each activity is translated as a call to a predicate. A variable assignment v `<-`e generates a goal that evaluates e in η and unifies its result with variable X; the remaining activities R are translated with η updated with the new binding $[X/v]$. `Invoke`

[2] Following Prolog notation an empty list is written [] and a list with head A and tail R is written $[A|R]$.

```
<sequence>
  <while name='a_13'>
    <condition>$i>0</condition>
    <scope>
      <assign name='a_14'>
        <copy><from>$i - 1</from><to variable='i'/></copy>
      </assign>
      <assign name='a_15'>
        <copy><from>$resp.body/factory:part[$i]</from>
          <to variable='p'/></copy>
      </assign>
      <invoke name='a_16' portType='factory:sales'
        operation='cancelReservation' inputVariable='p'
        outputVariable='r'/>
    </scope>
  </while>
  <throw faultName='factory:unableToCompleteRequest'/>
</sequence>
```

(a) A BPEL code fragment

```
while( '$i>0', (                                    % a_13
  '$i' <- '$i␣-␣1',                                 % a_14
  '$p' <- '$resp.body/factory:part[$i]',            % a_15
  invoke( factory:sales,cancelReservation,'$p','$r') % a_16
)),
throw( factory:unableToCompleteRequest)
```

(b) The intermediate representation.

```
a_13(A,B,C,D,E):- % ($i,$p,$resp.body/factory:part,$r,Y)
       A>0, !, a_14(A,B,C,D,E).
a_13(A,B,C,D,E):-
       E=fault('factory->unableToCompleteRequest').

a_14(A,B,C,D,E):-
       F is A-1, a_15(F,B,C,D,E).

a_15(A,B,C,D,E):-
       nth(A,C,F), a_16(A,F,C,D,E).

a_16(A,B,C,D,E):-
       'service_factory->sales->cancelReservation'(B,F),
       (  F=fault(G) -> E=fault(G)
       ;  F=reply(H) -> a_13(A,B,C,H,E)).
```

(c) Translation into logic program.

Figure 7: Translation example.

is similar, but it calls the target service predicate to obtain the result. `if` and `while` encode their condition with a call to a predicate C and a cut.

A `scope` is translated by nesting the translation of the activity/fault handler A'_H within updated environment $\eta[D]$, followed by a check for completion or faults. Faults within the scope are handled by H, and outgoing faults are rethrown. `flow` is translated similarly to `scope`, but without actually parallelizing the execution, since we are interested in the computational cost of the `flow` regardless of the number of threads. Links are modeled as Boolean variables, and dependent activities are sequenced to respect conditions on incoming/outgoing links. Dead-path elimination is supported.

3.5 A Translation Example

A translation example is presented in Fig. 7. Subfigure (a) is a BPEL fragment of an orchestration, (b) is the corresponding intermediate form, and (c) is the translation into a logic program. The orchestration traverses the list of part types to reserve from the external part

Resource	**With fault handling**		**Without fault handling**	
($n \geq 0$: input arg. value)	lower bound	upper bound	lower bound	upper bound
Basic activities	2	$7 \times n$	$5 \times n + 2$	$5 \times n + 2$
Single reservations	0	n	n	n
Cancellations	0	$n - 1$	0	0

Table 4: Resource analysis results for the group reservation service.

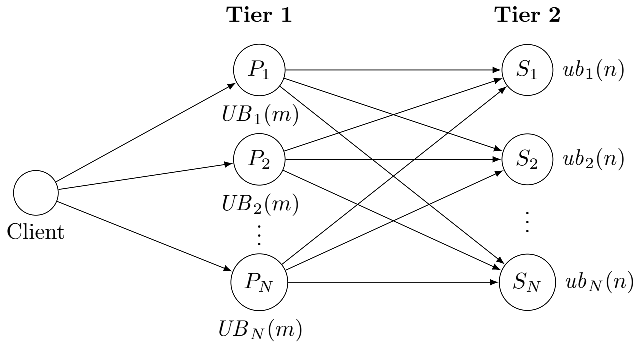


Figure 8: Two-tier simulation setting.

maker sales service.[3] If a fault arises, a fault handler tries to cancel already made reservations before signaling failure to the client. The figure shows just the `while` loop, which finishes with a `reply`.

The resource analysis finds out how many times external service invocations will be performed during process execution, from which deducing the number of messages exchanged is easy. The results for the complete orchestration are displayed in Table 4, where the estimated upper and lower bounds are expressed as a function of the input message.[4] We differentiate two cases: one in which fault-free execution is assumed, and another where fault handlers can be executed, which gives more cautious estimates. These two cases were obtained by turning on or off the generation of Prolog code for fault handling –the last part of Fig. 7 (c).

4 An Experiment in Adaptation

To validate our approach, we performed a simulation to study the effectiveness of applying data-aware computational cost functions to matchmaking and dynamic adaptation. We simulate a service network (Figure 8) where a client C selects among a set of providers P_i to reserve $n = 1..50$ sets of car parts. Each set consists of $M = 5$ different part types. The external client chooses one P_i which in turn chooses from among a set of part suppliers S_i, shared between all the providers. All P_i and S_i are known to be semantically equivalent, but vary in response time as the QoS attribute of interest. A P_i or S_i may fail with some probability p_f. When this happens, adaptation is triggered by searching for another (next-best) service from the pool.

The selection policies we have simulated are: random selection from the pool of candidates, fixed preferences, and data-dependent QoS prediction based on computational cost.

[3] Unlike in the example in Section 2.1, this code does not query different factories.

[4] The analyzer took 1.811 seconds to infer this information on a Intel Core Duo 2GHz machine with 2GB RAM and Darwin Kernel v10.2.0.

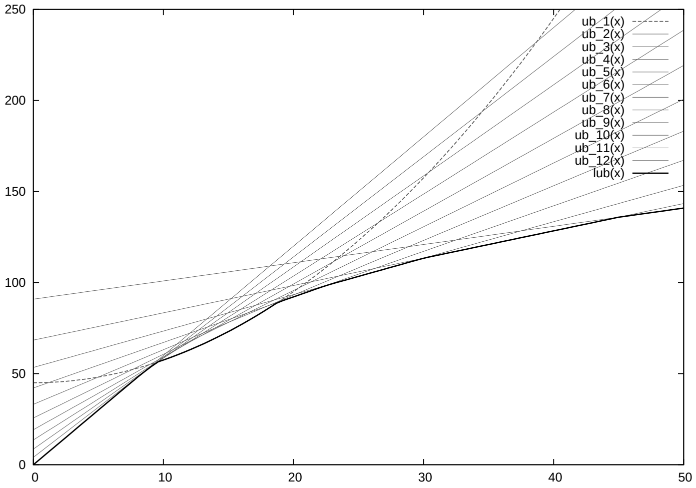


Figure 9: Upper bounds for computational costs.

In the last case, we select the best candidate taking into account its upper bound complexity (worst case behavior). Every service $S_i, 1 \leq i \leq 12 = N$, has a different upper bound cost function $ub_i(n)$ (portrayed in Figure 9), where n is the number of sets of a given part type. The bold line highlights the lowest upper bound among all the services for each n. $ub_i(n)$ measures the maximum number of messages exchanged by S_i as a function of the size of the incoming data. The computational cost for provider P_j is computed with the expression

$$UB_j(n) = E_{P_j}(n) + M + M \times ub^*(n)$$

which takes into account both the structural computational cost E_{P_j} (using the same family of curves in Figure 9) and that incurred by the services in the second layer: M times the cost ub_* of a service S_* selected for given n under the given selection policy.

Message exchanges are assigned a fixed time to convert them into execution time.[5] In a real scenario, this fixed amount of time can be updated as execution proceeds to reflect e.g. network state or system load.

The fixed preferences policy ranks services using the expected response time for some representative input; we chose $n = 12$. Therefore all queries whose data size is 12 are handled equally by both the fixed preferences and the data-dependent complexity cost approaches.

For each selection policy and for each n in the range 1..50, one hundred simulations are run and averaged. Each run performs matchmaking and simulates the execution of the selected service. Besides failures, the simulated number of outgoing messages in the run is (uniformly) randomly chosen between 60% and 100% of the upper bound, to model that sometimes this upper bound may not be needed. The time associated with every message exchange is padded with additional noise having a normal distribution to simulate the variations in the behavior of

[5] We are not taking into account the time associated to executing internal activities. The same technique used to infer the number of messages can be used to infer the number of activities of every type associated to some invocation, and can be accounted for in similarly to messages.

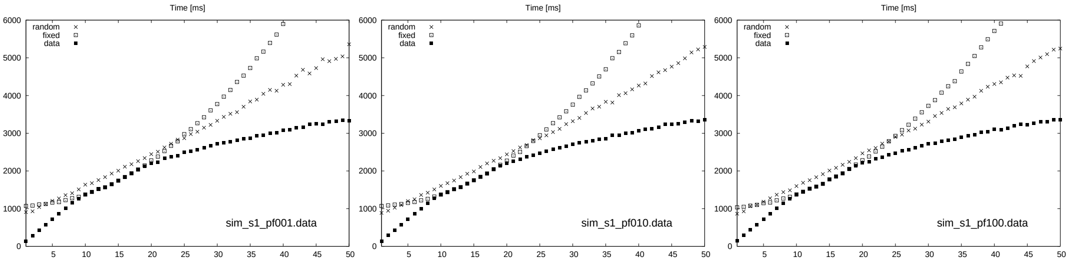


Figure 10: Simulation results for $\mathbf{p_f = 0.001, 0.01, 0.1}$ (left to right) and same noise distribution.

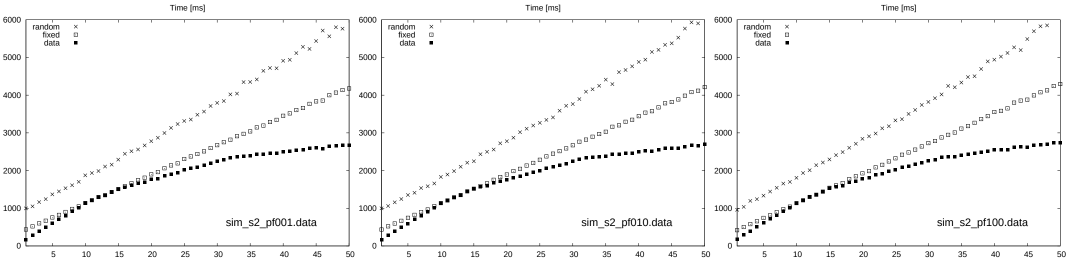


Figure 11: Simulation results for $\mathbf{p_f = 0.001, 0.01, 0.1}$ (left to right) and different noise distribution for each service.

the network.

Several sets of simulations with different time noise distribution parameters were performed, of which we have chosen two representative ones. In Fig. 10 all services have the same per-message average time (5 ms). In Fig. 11, services in both layers are assigned a different time per message whose average is in the range 4-8 ms. The figures show plots for the three selection policies, per each of the three failure probabilities used (left to right).

For most values of n, the data-dependent selection policy gives the best results and, notably, they feature a homogeneous and predictable behavior w.r.t. failure rates $p_f \in \{0.001, 0.01, 0.1\}$ and timing noise. Of course, it coincides with the selection made using the fixed preference policy for $n = 12$, where the fixed preferences were calculated. In an extended set of simulations (not appearing in this paper due to space constraints), the same behavior appears for even higher failure rates.

5 Conclusions

We proposed using data-aware computational cost functions to predict QoS adaptations and presented some preliminary results. We developed a translation-based scheme which, from an orchestration (in BPEL+WSDL), generates a (logic) program that can be analyzed by existing tools to automatically derive functions which are the upper and lower bounds of its computational cost. These functions are used to build more precise QoS estimations taking data characteristics into account which, in turn, can be used to, e.g., perform more precise predictive monitoring and proactive adaptation. We have reported on the results of a series of simulations where such data-aware QoS estimations were used to improve the efficiency of dynamic, run-time adaptation. The results are promising in that the data-aware adaptation always performs as well as any of the other policies studied, and in general gives better better results, even for cases with a very large variability in service behavior.

References

1. Mohammad Alrifai and Thomass Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *International World Wide Web Conference*, pages 881–890. ACM, April 2009.

2. J. Cardoso. About the Data-Flow Complexity of Web Processes. In *Int'l. WS on Business Process Modeling, Development, and Support: Business Processes and Support Systems: Design for Flexibility*, pages 67–74, 2005.

3. J. Cardoso. Complexity analysis of BPEL web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

4. D. Jordan et al. Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, et. al., 2007.

5. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

6. M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

7. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Int'l. Conf. on Logic Programming*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.

8. Wil van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, 2006.

Taming Dynamically Adaptive Systems Using Models and Aspects*

Brice Morin[1], Olivier Barais[2], Grégory Nain[1] and Jean-Marc Jézéquel[1,2]
[1]INRIA, Centre Rennes - Bretagne Atlantique
[2]IRISA, Université de Rennes1
Campus de Beaulieu
35042 Rennes Cedex - FRANCE
{Brice.Morin | Gregory.Nain}@inria.fr
{Olivier.Barais | Jean-Marc.Jezequel}@irisa.fr

Abstract

Since software systems need to be continuously available under varying conditions, their ability to evolve at runtime is increasingly seen as one key issue. Modern programming frameworks already provide support for dynamic adaptations. However the high-variability of features in Dynamic Adaptive Systems (DAS) introduces an explosion of possible runtime system configurations (often called modes) and mode transitions. Designing these configurations and their transitions is tedious and error-prone, making the system feature evolution difficult. While Aspect-Oriented Modeling (AOM) was introduced to improve the modularity of software, this paper presents how an AOM approach can be used to tame the combinatorial explosion of DAS modes. Using AOM techniques, we derive a wide range of modes by weaving aspects into an explicit model reflecting the runtime system. We use these generated modes to automatically adapt the system. We validate our approach on an adaptive middleware for home-automation currently deployed in Rennes metropolis.

1 Introduction

Society's increasing dependence on software-intensive systems is driving the need for dependable, robust, continuously available adaptive systems. Such systems often propose many variability dimensions with many possible variants, leading to a wide number of possible configurations that is difficult to integrally check at design-time because of time and resource constraints. For example, associations and public institutions of the metropolis of Rennes are working together on a project which aims at allowing dependent people to stay at home as long as possible. Due to the large scale of the project, and the diversity of disabilities that have to be considered, the deployment context will be different for each equipped house. Furthermore each deployment context is going to continuously evolve along with the evolution of the person's disabilities.

This ability to evolve a system at runtime is one critical aspect of achieving continuously availability. Many popular programming frameworks such as OSGI [33] or Fractal [7] now provide support for dynamic adaptation through extension mechanism such as plugins or variability mechanism through introspection and reconfiguration API. However the high variability of crosscutting and non-crosscutting features in adaptive systems introduces an explosion of possible runtime system configurations (often called modes). When new features are introduced at deployment time (vs. design time), we also have to make sure that they do not lead the system into unwanted modes. Besides, due to the fact that features are often partially independent, the (implicit) state-machine representing the path between modes is highly connected, leading to a nearly quadratic explosion of transitions between modes [5, 35]. The inefficacy of the variability and extension mechanisms to tame the high-number of modes transitions might lead to several undesirable consequences related to Dynamically Adaptive System maintainability, including partial duplication of reconfiguration scripts or the non-cover of all the modes transition, etc.

Aspect-Oriented Modeling (AOM) was initially introduced to improve the modularity of software [11, 21], complementary to Model Driven Engineering (MDE)

*This work was partially funded by the DiVA FP7 European project (STREP) (See http://www.ict-diva.eu/) and the S-Cube European Network Of Excellence on Software Services and Systems (See http://www.s-cube-network.eu/).

to link models to the real world [14]. In [24], we have proposed a first approach that leverages AOM and MDE to manage variability at runtime. It relies on the notion of aspect models, that can be woven into an explicit model of the runtime configuration seating on top of the running system. Actual mode switches between runtime configurations are then triggered by reconfiguration scripts automatically generated based on the differences between the initial model and the newly woven one. The result of this approach is that modes becomes somehow implicit from the point of view of the system designer, and new modes can appear when new aspects are introduced during the life of the system. It is thus no longer possible to statically validate every accessible mode and each mode transition.

In this paper, we show how aspects can help designers determine interactions between dynamic variants and how runtime models can be used to validate new configurations on the fly, before committing them on the running system (making it easy to roll-back when a configuration is not valid). Indeed, once the system has been deployed, new variability dimensions and variants that have not been foreseen may appear while the system is running and cannot be stopped. In this case, it is very useful to validate configurations on the fly before actually adapting the running system.

The reminder of this paper is organized as follows. Section 2 describes the general process for managing dynamic variability. Section 3 presents how we leverage AOM and MDE techniques and tools to determine interactions and validate configurations on the fly. Section 4 outlines how our approach was validated in a home-automation context deployed in the metropolis of Rennes. Section 5 discusses related work and section 6 concludes by presenting a set of open research problems based on our experience.

2 Process Overview

This Section presents our approach for managing variability at runtime [24]. The overall process is described in Figure 1.

2.1 Maintaining a High-Level Representation of the Running System

Maintaining a model at runtime [4] representing the running system (Figure 1, step 1) allows us to reason on the model and manipulate the model independently from the running system. Using high-level abstractions, we can discard all the platform-specific runtime information we do not need and reason more easily and more efficiently in the next steps of the process.

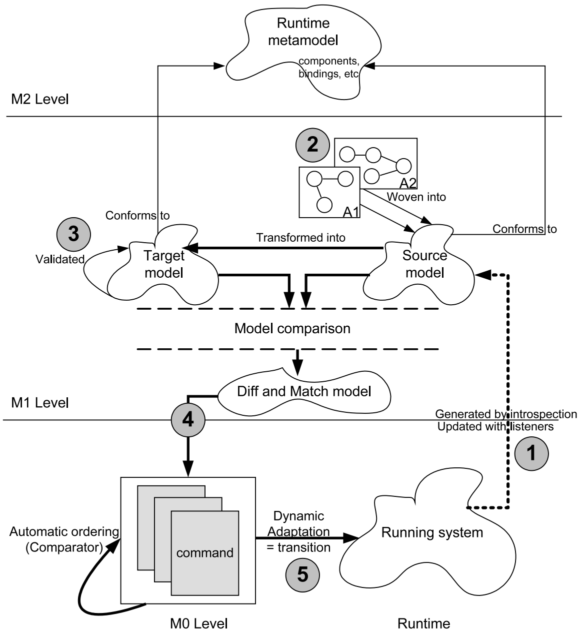


Figure 1. MDE and AOM for Dynamic Adaptation

Recent component-based middleware platforms like OSGi [33] propose introspection APIs that allows discovering the architecture of a running system. We use these APIs to collect and format relevant information in the form of a platform-independent and high-level model. In the case of large systems, using introspection in order to generate a reference model from scratch may be time-consuming especially when only small changes appear. To tackle this issue, we observe the architectural reconfigurations appearing in the running system in order to update the model. This limits the flow of data manipulated in the system. Moreover, recent middleware platforms already propose this kind of observers or propose mechanisms to easily implement such kind of observers [6, 7]. Note that the changes that may affect the source model are not directly reflected to the running system.

2.2 On-demand Construction of Configurations with Aspects

In order to manage variability and avoid the combinatorial explosion of artifacts needed to support this variability, we propose to focus on variation points and variants instead of focusing on whole configurations. A variability dimension is a particular concern that may be realized in different ways. We use aspects to represent the different variants of a variation point. Using Aspect-Oriented weavers, whole configurations can be built on-demand by selecting a set of aspects as illustrated in Figure 1, step 2. In practice, we use SmartAdapters [18, 25] an Aspect-Oriented Modeling (AOM) tool for weaving aspects at a model level. However, the approach presented in this paper is not dependent from SmartAdapters and other AOM tools like MATA [13] could also be used. Components present in all the configurations constitute a base model where aspects are woven.

SmartAdapters has formerly been applied to Java programs and UML class diagrams [18]. More recently, we have generalized this approach to any domain-specific modeling language [23]. This allows us to leverage the notion of aspect for runtime models representing at a high level of abstraction the architecture of a system at runtime. SmartAdapters automatically generates an extensible Aspect-Oriented Modeling framework specific to our metamodel.

In SmartAdapters, an aspect is composed of three parts: *i)* an advice model, representing what we want to weave, *ii)* a pointcut model, representing where we want to weave the aspect and *iii)* weaving directives specifying how to weave the advice model at the join points matching the pointcut model. The advice model is a model fragment representing a given concern. In our case, it represents a pre-assemby of components that may not be fully specified. The pointcut model is also a model fragment that is parameterized by roles (See [31]), equivalent to wildcards in AspectJ pointcuts [15, 16]. Both the advice model and the pointcut model are defined using the concrete syntax of the domain. Finally, weaving directives specify how to integrate the advice model into the target model, using a generated domain-specific action language [23].

We (optionally) extend each aspect with a context describing when to trigger the weaving of aspects. A context is a slice of the environment describing when the aspect is useful and its impact on QoS properties. For example, a buffering aspect can *optimize* the bandwidth of the network if the system has enough free memory (*e.g.* > 512 Mb) and if the bandwidth is saturated (*e.g.* > 90%). Aspects with a context are chosen according to the execution context and the QoS properties to optimize [12]. Aspects with no context can be manually triggered by the user.

Figure 2 illustrates an internalization (`I18N` for short) aspect. The pilot of the case study is currently deployed in Rennes. Rennes is an international city hosting many students from different countries where lots of different languages are spoken. Within a single day, people from different countries may transit in the home. Systematically translating all the information in all the possible languages may cause an information overhead that could make information difficult to catch. This is why internationalization should be handled dynamically.

In order to design this aspect, we leverage the ability of SmartAdapters to integrate variability into aspects [18]. Indeed, each language (`EN`, `FR`, `DE` in Figure 2) is considered as a variant. The behavior of the advice can be described as follows. The `I18N` interface provides a set of methods responsible for translating a pre-defined set of labels. It also provides a look-up method `get(String myString): String` that returns, if possible, the translation of `myString` for a given language. In a component (*e.g.*, `FR`), if the look-up method cannot translate a word, the component ask the dispatcher to find a translation for this word. The dispatcher will ask the components according to a predefined policy (*e.g.*, `EN` first if available). If the word exists in the local database, it is translated (*e.g.*, from English to French) thanks to the translator that sends a request to a website dedicated to translation. Note that in the advice, all the components and bindings are unique (depicted with a '1' in the Figure). This means that even if there exist multiple join points and/or even if the aspect is applied several times, these elements will only be woven once in the base model. In the composition protocol, the bindings are not unique and will be introduced for all the identified join points. The pointcut simply identifies any component that requires the `I18N` interface, with no more assumption. The weaving process has 2 steps: matching (or join point detection) [31] and composition. In our example, the matching step detects all the components that require the `I18N` interface. Then, the composition step binds the `I18N` server interface provided by the advice to the client interfaces of the join points. When an aspect is being composed, the inverse composition protocol is automatically generated. It allows us to easily unweave an aspect when it is not longer adapted to the context.

After some aspects have been woven into the source model, the target model we obtain is validated, as illustrated in Figure 1, step 3. We will present in more details this validation step in Section 3.

2.3 On-the-fly Generation of Reconfiguration Scripts

As mentioned by Zhang and Cheng in [35], if there exists N possible configurations, this may lead to N(N-1) possible transitions. If N is large, it rapidly becomes difficult to handle these transitions by hand.

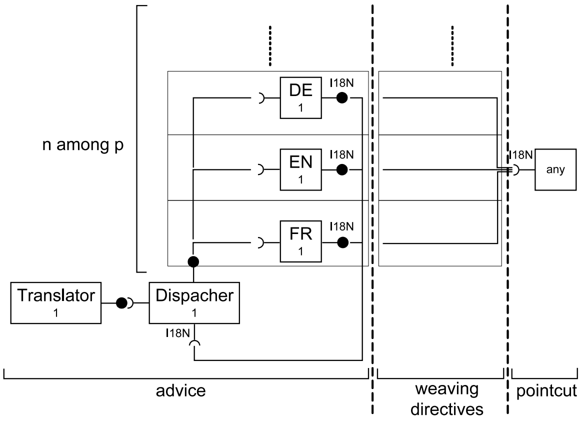


Figure 2. I18N aspect

In a traditional model-driven development, models are refined and transformed step by step down to source code. If some changes appears in the requirements, it is possible to propagate these changes to the models and regenerate the source code in order to rapidly propose a new version of the system. In the context of adaptive systems, it is not possible to adopt this schema. Indeed, such systems should offer continuous services and cannot be stopped, regenerated and restarted. The system should keep executing while being reconfigured from one configuration to another.

Once a target model (representing the system we want to reach) is created and validated, it is compared with the source model (representing the actual architecture of the running system). In the current implementation of our tool, we use EMF Compare[1] in order to compare models. It produces a diff and a match model that specifies the differences and the similarities between the source and the target models, as illustrated in Figure 1, step 4. The comparison engine is generic, so it is possible to compare any kind of models. The algorithm is quite similar to the algorithm proposed by Nejati *et al.* in the context of statechart specifications [28]. It considers the properties of each model element as well as its neighbors in order to compute a similarity degree. Note that it is possible to customize the comparison engine to consider the specificity of a given domain metamodel. However, the generic engine provides sensible results for our metamodel describing runtime architecture, with no customization.

Then, we automatically analyse both diff and match models to obtain the relevant changes between the source model and the target model *e.g.*, addition/removal of components/bindings, changes of attribute values, etc. However, it is not possible to adapt the running system during this analysis. For example, if the model comparison detects a component removal before a binding removal, directly adapting the system would lead to a dangling binding that might not be allowed by the underlying execution platform.

In order to tackle this issue, we reify each significant modification as a reconfiguration command during the analysis (Figure 1, step 4). Each command implements an atomic platform-specific reconfiguration (adding and/or removing bindings and/or components, etc.) and declares a priority. We first stop the components that have to be stopped and then remove bindings before removing components. We add components before adding bindings and finally restart the components that should be restarted. When the analysis of the model comparison is achieved, we execute the ordered sequence of commands to actually adapt the running system in a safe way (Figure 1, step 5). This set of commands is the transition that transforms the source system into the target system. Depending on the execution platform we use (*e.g.*, OSGi, Fractal or OpenCOM) a factory will instantiate the corresponding commands.

3 Validating Target Configurations

In Section 2, we showed how we can obtain configurations by weaving some aspects into a base configuration. Using aspect models, instead of directly adapting the running system using low-level reconfiguration scripts [10] allows us to reason more easily and help designers in identifying interactions between aspects [13]. More details about aspect interaction detection are given in Section 3.1.

Then, we showed how to generate reconfiguration scripts to make the running system evolve from a source configuration (the current configuration), to a target configuration. However, in the context of adaptive systems, we should ensure that the target configuration we want to reach makes sense. This is why we do not directly reflect the changes appearing in the source model to the running system. When the number of configurations is limited, for example in the case of critical embedded systems, it is possible to validate at design time all the possible configurations [35]. However, in larger-scale adaptive systems, this systematic validation may become too time and resource consuming to be realistic. Moreover, once the system has been deployed, new variation points that have not been foreseen may appear while the system is running and cannot be stopped. In this case, it is very useful to validate configurations on the fly before actually adapting the

[1] See http://www.eclipse.org/modeling/emft/

running system. This is detailed in Section 3.2.

3.1 Detecting Aspect Interactions

In Section 2 we introduced an internationalization aspect. Most of the aspects of our case study (see Section 4) and most of the components of the base system (for example, GUI) also needed this aspect. Weaving the I18N aspect before the other aspects may cause some important messages not to be translated. Using techniques like Critical Pair Analysis (CPA) allows us to detect interaction between aspects [13]. Basically, if the pointcut of an aspect *A1* can be matched in the advice of another aspect *A2* it means that *A2* introduces some join points for *A1*. In other words, *A1* should be woven after *A2* in order to be able to consider newly introduced join points.

For example, let us introduce a second aspect responsible for preventing devices deployed in the house (lights, electric shutters, etc) to be damaged due to too many successive transitional regimes. This aspect, called **event filter**, will ignore all the antagonist actions that appears in a too short period. All the events sent to the unstable device controller by the device proxy are derived into the event filter component. These events are cached during a given period that depends on the type of the device. Events are delegated to the device if no antagonist events appeared during the cache period. These filters cannot be systematically deployed as they make devices less reactive in standard conditions. This is why filters should be dynamically and locally deployed and undeployed. Every canceled actions has to be logged and displayed in a language the user understand, using the I18N interface. As the event filter aspect introduces a component that requires the I18N interface, it introduces a new join point for the internationalization aspect. Consequently the I18N aspect should be woven after the event filter aspect.

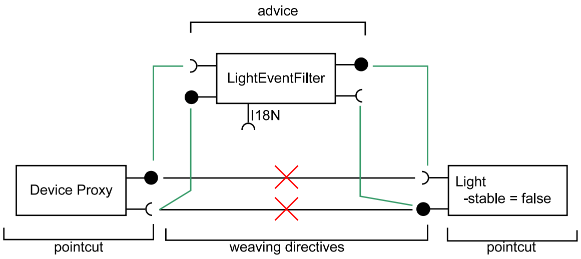


Figure 3. Event Filter aspect

This kind of basic analysis can help designers in identifying interactions that cannot easily be detected when directly working with low-level reconfiguration scripts. However, CPA has limitations. For example, this kind analysis is not associative. If no interaction exists between *A1* and *A2* and no interaction exists between *A1* and *A3*, nothing ensures that there exists no interaction in the triplet $\{A1, A2, A3\}$, depending on the weaving order. Determining the interactions that may exist in all the possible subsets of aspects is very complex as the number of combinations grows rapidly.

3.2 Validating Target Configurations

As explained in the introduction of this section, it is not always possible to check all the possible configurations of an adaptive system *a priori*, for time and resource issues. Moreover, the apparition of unforeseen adaptation while the system is already deployed makes it impossible to perform all the validation process at runtime. However, in the context of high-insurance adaptive systems [20] any configuration we wanted to reach should be validated.

In order to validate target configurations (Figure 1, step 3), we propose to define some invariants on the metamodel we use to represent runtime architecture and check these invariants for every constructed (by aspect weaving) target configuration. These invariants are expressed as Kermeta [26] meta-aspects that are woven into the metamodel we are using for representing runtime architecture. Kermeta meta-aspects can be used to refine existing meta-classes by integrating contracts (pre/post-conditions, invariants), attributes and references, operations and super-classes. The invariant illustrated in Figure 4 specifies that all the client and non optional ports defined in the component type of the component should be bound. In other words, it detects if mandatory bindings are missing. It uses an OCL-like syntax[2] which provides high-level operators for navigating and querying models: *select*, *exist* and *forall* in our example. Another invariant checks that the server interface of a binding is a sub-type of the client interface. Currently, six invariants are woven into our metamodel. Note that end-user can define more specific invariants to ensure the validity of the configurations.

Invariant checking, as well as the steps related to aspect weaving and aspect interaction detection, can be performed on a tiers system, independent from the running system itself. Indeed, all the models (metamodel, configurations and aspects) can be serialized in XML and transmitted to other systems.

[2] Kermeta now allows to define constraints using the real OCL syntax

```
aspect class Component {
  inv optionalClientPortBound is do
    self.type.ports.select{p |
        not p.isOptional and
        p.role == PortRole.CLIENT
    }.forAll{p |
        self.binding.exists{b |
          b.client == p
        }
    }
  end
}
```

Figure 4. Checking mandatory bindings

Another possible solution to validate target configuration before actually adapting the running system would be to simulate models. This can be done by describing the behavior of each configurations, for example using state machines or Petri nets [35]. Then, it would be possible to use Kermeta [26] to execute these models and perform the simulation and detect deadlocks, for example. However, in order to manage the explosion of variants, this behavior should be associated to each aspect and should be composable in order to obtain the behavior of whole configurations. SmartAdapters is well adapted to compose structural aspects, in class or component diagrams. However, to consider the semantic of behavioral models, it has to be customized by hand. Another solution would be to use a specific aspect weaver dedicated to the composition of behavioral models. Such an approach is presented in Section 5 and its integration with our approach is discussed in perspectives.

If the target configuration we want to reach is valid, then the process continues, as illustrated in Figure 1, step 4, in order to actually adapt the running system. If the target configuration is not valid, then our rollback mechanism simply consists in discarding this target configuration and do not submit it to the following steps of the process. Indeed, as the modification on the model are not directly reflected to the running system, we do not have to cancel platform-level reconfigurations. An error report is automatically raised by Kermeta [26], specifying which invariants are violated by the target configuration. This helps the system or the user in understanding why the configuration is not valid. For example, if we consider that the I18N client port is mandatory, then we are able to detect the cases where the I18N has been woven before (or not at all) other aspects, without using the preliminary critical pair analysis. Indeed, the invariant illustrated in Figure 4 detects missing mandatory bindings. This case is illustrated in Figure 5 that shows a fragment of the base model where the event filter has been woven and the I18N is not woven at all.

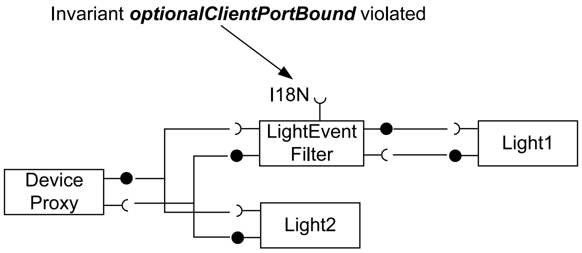


Figure 5. Invariant violated

4 Application to Home-Automation

4.1 EnTiMid to help people to stay at home

Industrials, associations and public institutions of the metropolis of Rennes, are working together on a project which aims to allow dependent people to stay at home as long as possible. Due to the large scale of the project, and the diversity of disabilities that have to be considered, the deployment context will be different for each equipped house. The technologies used will vary, in order to compensate handicaps or because a technology is already installed, and people do not want it to be removed. Moreover, the system installed in these houses will have to provide a remote access to the devices of the house, and transmit all the necessary information from the sensors of the house to a control center where information will be treated. Those access and transmissions can be realized through various ways (Internet, POTS, SMS) and the medium used will vary according to the availabilities.

An abstraction layer over all these devices has been developed in the form of a multi-facet middleware called EntiMid [27]. Based on an OSGi platform [33], EnTiMid is composed of different components (called bundles), that can be dynamically added, removed, started or stopped, with no need to restart the entire system. Each of them can offer or require services to/from other ones, but such services can disappear at any time. One of those services, identified as *Tech-Provider*, aims at translating the information caught from a device network protocol, into EnTiMid standard event messages, and vice versa. By this way, high level services can manage devices through unified messages, whatever the underlying technology is. EntiMid allows engineers to prescribe the most adapted technology, with no regard to the communication technology it uses. Then high level services can be developed to offer different kind of services to inhabitants and health professionals. For example, automatic energy, heating, access or light management to ease the everyday life; remote control and alert transmissions, to allow professionals to intervene on the house, in a short time, according to the information collected.

A show apartment will soon be available, and EnTiMid will be deployed in order to test its functionalities with devices installed by industrials. Those real conditions will have for consequence an identification of new needs of development and variability.

4.2 Designing Variability Dimensions

We now introduce and justify the need for dynamic variability in our case study. Each variability dimension is represented by a set of aspects designed with SmartAdapters.

Device Management. Physical devices are managed by EnTiMid. Most of the devices are installed when the system is deployed. However, new physical devices may be installed after the initial deployment and consequently, they should be managed dynamically while maintaining the functionalities offered by already deployed devices. In our case study, we have to manage 6 lights, 4 heaters, 3 mixing valves (controlling water temperature) and 3 electric shutters. Moreover, depending on the evolution of the patient, devices should be managed in different ways. For example, in the case the patient becomes visually impaired, the power of the lights should be increased by 10%. Another focus can be the evolution of a mobility handicap. In a first stage, the patient can move alone in the house, allowing him to manually close the shutters. Then the evolution of the disease makes it difficult for the patient to get out of his bed. A remote control will then be offered to this person to ease his everyday life, and the control system of the house have to adapt to this situation.

Each low-level protocol (KNX[3], X10, X2D[4], etc) manages devices in an ad-hoc way. Consequently, we would have to define a variability dimension for each protocol. For the sake of clarity, we present one generic variability dimension that harmonizes the concepts present in each low-level protocol. It is illustrated in Figure 6. In order to represent this variability dimension, we leverage the ability of SmartAdapters to integrate variability into aspects [18]. Each type of device controller (Light, Heater, Shutter and Mixing Valve) is a variant. Each type of component can be instantiated several times. Each device also offers an interface for loading pre-defined scenarios. All the device controllers are connected to a device proxy that receives messages from the EnTiMid platform and dispatches these messages to the appropriate device.

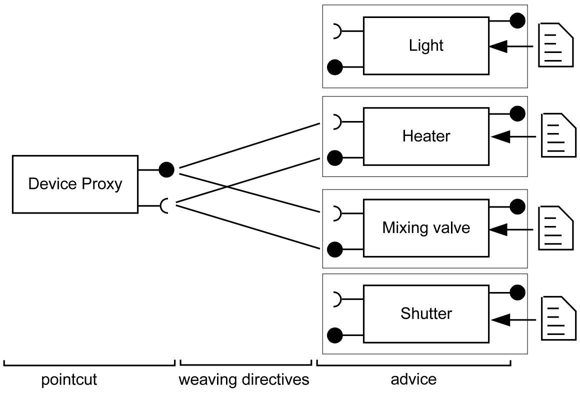
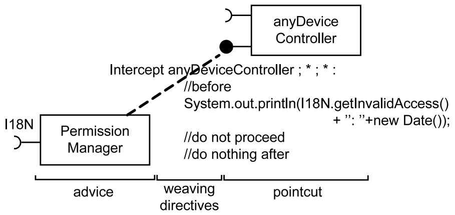


Figure 6. Device Management aspect

Permission Management. All the physical devices in the house are supposed to be potentially controlled by EnTiMid. However, doctors can choose for each device whether it is controlled by the system or by the patient, according to the degree of autonomy of the patient. If the device has no permission manager, the patient can interact with the device. With a permission manager, the patient cannot interact with the device that only follows a pre-defined scenario.

The permission management aspect illustrated in Figure 7. In order to control the access from the user, an aspectual component [29] intercepts, using an AspectJ-like pointcut, every call to the services of the controller, log each attempt and does not proceed. The device will simply execute its pre-defined scenario without being interrupted.



```
Intercept anyDeviceController ; * ; * :
        //before
        System.out.println(I18N.getInvalidAccess()
                                    + ": "+new Date());
        //do not proceed
        //do nothing after
```

Figure 7. Permission Management aspect

Two other aspects (internationalization and event filter) have already been introduced in previous sections. Table 1 summarizes the number of aspects we

[3] http://www.knx.org/

[4] http://www.english.deltadore.com

really need for each variability dimension to implement our case study.

Dimensions	Aspects	Aspect variants
Device Mgmt	3 (KNX, X10, X2D)	4 (1 per device type)
Permission Mgmt	1	0
Event Filter	3	4
I18N	1	10
Total	8	18

Table 1. Number of aspects per variability dimension

4.3 Constructing a Configuration by Aspect Weaving

We are now going to propose to illustrate the weaving process with some of the aspects we have identified in the previous sub-section into our motivating example (Figure 8). The top of the figure illustrates a snippet of the base configuration. It allows to access low-level devices using two high-level protocols: UPnP[5] and DPWS[6], via the EnTiMid component. In the bottom part of the figure three aspects have be woven: a filter aspect that filters the events send to Light1 ; two permission managers that restrict the heater and the shutter to their pre-defined scenarios. Finally, the internationalization aspect is woven. The interaction between the aspects can be observed in the example: the event filter and the permission manager aspects uses the internationalization aspects.

With the five variability dimensions we have defined earlier in this section, we can obtain a wide range of possible configurations. If we consider the three aspects that directly impact devices (device management, permission management and event filter), we obtain five possible modes for each device, as shown in Table 2, where 0 indicate that the aspect is not active for the device. As the devices are managed independently, this leads to $5^{16} \approx 15 \cdot 10^{10}$ possible configurations. This number is even greater if we consider the internationalization aspect.

If we consider the $15 \cdot 10^{10}$ possible configurations, we obtain approximately $225 \cdot 10^{20}$ possible transitions from one configuration to another. The configurations are obtained on-demand: the weaving of some aspects can be triggered by hand depending on the choices of a human operator (doctor or technician), while some other aspects (*e.g.*, event filter) are triggered by the context. Before adapting the system, all the invariants defined in the metamodel are checked on the woven target model. This allows us to determine whether the target model is well-formed or not. If the configuration is valid, the transition toward the target model is automatically computed using model comparison. If we consider the internationalization aspect, we should multiply the number of configurations by 2^{10} as we should handle at least one language among 10. Table 3 summarizes the number of dimensions we have defined, the total number of aspects we need, the number of configurations we obtain on-demand by aspect weaving and the number of transitions we generate on-demand when moving from one configuration to another.

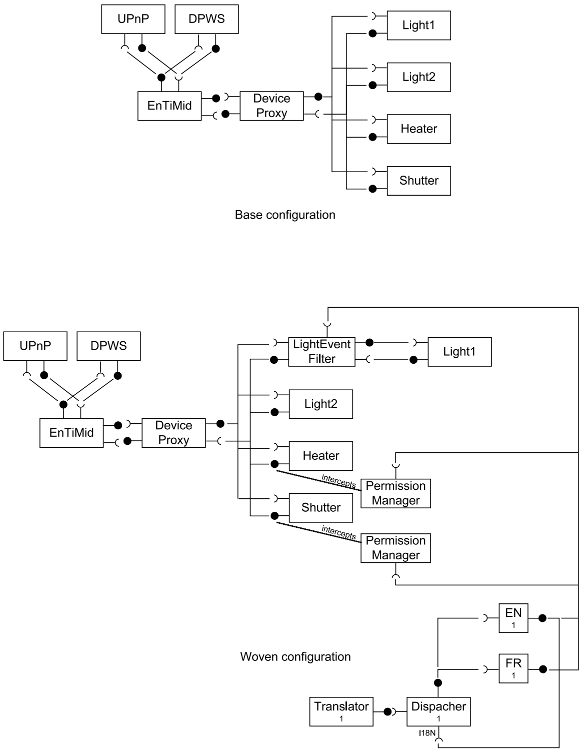


Figure 8. Base and woven configurations

Device Mgmt.	Permission Mgmt.	Event Filter
0	0	0
1	0	0
1	0	1
1	1	0
1	1	1

Table 2. Five Operating Modes per Device

[5] http://www.upnp.org/

[6] http://en.wikipedia.org/wiki/Devices_Profile_for_Web_Services

Dimensions	Aspects	Configurations	Transitions
4	8	$> 15 \cdot 10^{13}$	$> 225 \cdot 10^{26}$

Table 3. Number of configurations and transitions managed by aspects

5 Related Work

In [35], Zhang and Cheng propose to adopt a model-driven approach for designing and validating dynamically adaptive software systems. This approach focus on the behavior of adaptive systems whereas we mainly focus on the architecture of running systems. In [35], the behavior is modeled with state-based diagrams like Petri nets. Zhang and Cheng define an adaptive system as a set of simple adaptive systems. A simple adaptive system is defined by three entities: a source system, a target system and transitions responsible for moving the source system to the target system. All the source systems, target systems and transitions should be explicitly modeled, leading to an explosion of artifacts needed to manage an adaptive system. However, this exhaustive representation allows validating intensively the system at design time. Finally, using code generation, adaptive programs are derived from the models. In our approach, configurations are constructed on demand by selecting and weaving a set of aspects. Once composed, it is possible to check the target configurations we want to reach. Then, the transitions needed to adapt the system is automatically generated using model comparison.

In [10], David *et al.* present SAFRAN (Self-Adaptive FRActal compoNents) an Aspect-Oriented approach for implementing self-adaptive system on the Fractal [7] platform. In order to adapt the system, they define adaptation policies, separately from the business logic, which follow this pattern: **when** <*event*> **if** <*condition*> **do** <*action*> where actions are low-level reconfiguration scripts. Batista *et al.* [3] propose the same kind of approach for the OpenCOM [6] execution platform. These approaches do not propose an explicit representation of the target configurations. Consequently, it is not possible to easily visualize the system nor to perform validation, simulation before actual adaptation. In our approach, configurations are obtained by weaving aspects into a model representing the current system. Woven configurations are checked against invariants. We then generate all the adaptation logic needed to adapt the system while these approach have to specify low-level reconfiguration scripts. Moreover, our approach is not specific to a given execution platform. Finally, script-based approach do no offer support for easily determining interactions between reconfiguration scripts while Aspect-Orientation provides some mechanisms [13].

Ensuring software correctness is an important issue and this is amplified when dealing with software variation . Correctness is even more important in Dynamically adaptive System where variation is handled at runtime. This issue has been addressed by the software product lines community [30, 32]. One of the main concerns for correctness in product lines is about the methods to be used in order to limit the number of tests to be performed for a family of products. Two issues are especially considered: the increase of work for the programmer and the time spent to perform them [8, 19]. These contributions mainly introduce formal methods in order to exploit the commonalities of a software family in order to achieve these issues. They rely on SAT solver [9] or more generally on model-checking [17] techniques in order to verify those tests. In our case, the main difference is the fact that verifications can only be done at runtime. New aspect can be designed and integrated, consequently unanticipated evolution can occur. Using MDE techniques allows software developer to apply his aspect on an abstraction of its runtime system to check its correctness.

In [24], we present a first approach that combines AOM and MDE in order to manage variability at runtime. In this paper, we show how aspects can help designer in determining interactions between dynamic variants and how models allows to validate new configurations independently from the running system and easily roll-back when a configuration is not valid.

In [34], Wolfinger *et al.* demonstrate the benefits of integrating Software Product Line techniques to manage the runtime reconfiguration and adaptation mechanisms on the .NET platform. Automatic runtime adaptations are attained by using the knowledge documented in variability models. As many authors [1, 2, 22] advocate that aspect-oriented software development (AOSD) is an effective technique to support feature variability, this approach is close to ours. Automatic runtime adaptations are attained by using the knowledge documented in variability models. However, they do not propose to do a preview of the running system at the model level to check its correctness.

In the domain of Aspect-Oriented Modeling, Nejati *et al.* [28] propose an approach for matching and merging statechart specifications. This approach would be useful for extending our approach with behavior, as mentioned in Section 3.2. If we describe the behavior of our aspects it would possible to merge this behavior with the base model in order to obtain the complete behavior. Describing the behavior of our aspects mainly consists in modeling the behavior of the interfaces of each component and compose these behavioral models when components are assembled. Once we obtain the global behavior, it is possible to reuse the concepts proposed by Zhang and Cheng [35] for validating the behavior of adaptive systems.

6 Conclusion

In this paper, we have presented our approach for managing the complexity of dynamically adaptive systems. This approach combines aspect-oriented and model-driven techniques in order to limit the number of artifacts needed to realize dynamic variability. Our aspect model weaver allows us to construct configurations on-demand by selecting, by hand or according to predefined conditions, a set of aspects. Using the woven configuration, it is possible to validate this configuration before actually adapting the running system. Using aspects instead of low-level reconfiguration scripts allows us to detect some interactions that can provide assistance when selecting the set of aspects to be woven. Then, target configurations obtained after aspect weaving are checked with respect to the invariant we have defined into our metamodel. If a target configuration is not valid, the roll-back mechanism simply consists in not submitting this target configuration to the sub-sequent steps of the adaptation process. If the configuration is valid, we generate the adaptation logic using model comparison. This allows us to automatically determine a safe transition to make the system evolve from a its current configuration to the target configuration.

In future works, we plan to extend our approach following different axis. Currently, we describe our systems according to their runtime architecture (components, bindings, etc). We will also consider the behavior of dynamically adaptive systems. This can be realized if we can modularize and compose the behavior of components. Thus, it would be possible to decompose the system behavior into aspects, as we do for the architecture. We plan to reuse the approach we have presented in the related work section to consider the behavior. Another axis is about the validation of target configurations. Currently, we ensure that the target configurations ensure the invariants defined in our metamodel. With the definition of the behavior, we would be able to perform simulation in order to detect some deadlocks, for example.

References

[1] V. Alves, P. Matos Jr, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. *Transactions on Aspect-Oriented Software Development IV*, 4640/2007, 2007.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: 28th International Conference on Software Engineering*, pages 122–131, Shanghai, China, 2006. ACM.

[3] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *EWSA'05: 2nd European Workshop on Software Architecture*, pages 1–17, Pisa, Italy, 2005.

[4] N. Bencomo, G. Blair, and R. France. Models@run.time (at MoDELS) workshops. www.comp.lancs.ac.uk/ bencomo/MRT/.

[5] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *ICSE'08: Formal Research Demonstrations Track*, Leipzig, Germany, 2008.

[6] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. Opencom v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, USA, 2004.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[8] M. Cohen, M. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, Portland, Maine, 2006. ACM.

[9] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *GPCE'06: 6th Int. Conf. on Generative Programming and Component Engineering*, pages 211–220, Portland, Oregon, USA, 2006. ACM.

[10] P. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th Int. Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, Vienna, Austria, 2006.

[11] E. Figueiredo, N. Cacho, C. SantAnna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE'08: 30th International Conference on Software Engineering*, pages 261–270, Leipzig, Germany, may 2008. ACM.

[12] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel. Modeling and Validating Dynamic Adaptation. In *3rd International Workshop on Models@Runtime (MODELS'08)*, Toulouse, France, oct 2008.

[13] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[14] J.-M. Jézéquel. Model Driven Design and Aspect Weaving. *SoSyM'08: Journal of Software and Systems Modeling*, 7(2):to appear, march 2008.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[17] T. Kishi, N. Noda, and T. Katayama. Design Verification for Product Line Development. In *SPLC'05: 9th International Software Product Lines Conference*, volume LNCS 3714, pages 150–161. Springer, 2005.

[18] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Nashville USA, Oct. 2007.

[19] M. Mannion and J. Cámara. Theorem Proving for Product Line Model Verification. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003*, volume LNCS 3014, pages 211–224. Springer, 2003.

[20] P. McKinley, B. H. C. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. Harnessing digital evolution. *Computer*, 41(1):54–63, 2008.

[21] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[22] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT'04/FSE-12: 12th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 127–136, Newport Beach, CA, USA, 2004. ACM.

[23] B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.

[24] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *MoDELS'08: 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[25] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jzquel. Managing Variability Complexity in Aspect-Oriented Modeling. In *MoDELS'08: 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[26] P. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer. Kermeta is available at: http://www.kermeta.org/.

[27] G. Nain, E. Daubert, O. Barais, and J. M. Jézéquel. Using MDE to Build a Schizonfrenic Middleware for Home/Building Automation. In *ServiceWave'08: Networked European Software & Services Initiative (NESSI) Conference*, Madrid, Spain, december 2008.

[28] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *ICSE'07: 29th International Conference on Software Engineering*, pages 54–64, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[29] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A Component-Based and Aspect-Oriented Model for Software Evolution. In *IJCAT'07: International Journal of Computer Applications in Technology, Special Issue on Concern-Oriented Software Evolution*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.

[30] K. Pohl and A. Metzger. Software Product Line Testing. *Commun. ACM*, 49(12):78–81, 2006.

[31] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, page 15, Nashville USA, Oct. 2007.

[32] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. In *International Workshop on Software Architectures for Product Families*, volume LNCS 1951, pages 146–157, Spain, 2001. Springer.

[33] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4.1, May 2007. http://www.osgi.org/Specifications/.

[34] R. Wolfinger, S. Reiter, D. Dhungana, P.Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *ICCBSS'08: 7th Int. Conf. on Composition-Based Software Systems*, pages 21 – 30, 2008.

[35] J. Zhang and B. H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *ICSE'06: 28th International Conference on Software Engineering*, pages 371–380, Shanghai, China, 2006. ACM Press.

A Framework for Proactive Self-Adaptation of Service-based Applications Based on Online Testing*

Julia Hielscher[1], Raman Kazhamiakin[2], Andreas Metzger[1] and Marco Pistore[2]

[1] SSE, University of Duisburg-Essen, Schützenbahn 70, 45117 Essen, Germany
{hielscher,metzger}@sse.uni-due.de
[2] FBK-Irst, via Sommarive 18, 38050, Trento, Italy
{raman,pistore}@fbk.eu

Abstract. Service-based applications have to continuously and dynamically self-adapt in order to timely react to changes in their context, as well as to efficiently accommodate for deviations from their expected functionality or quality of service. Currently, self-adaptation is triggered by monitoring events. Yet, monitoring only observes changes or deviations after they have occurred. Therefore, self-adaptation based on monitoring is reactive and thus often comes too late, e.g., when changes or deviations already have led to undesired consequences. In this paper we present the PROSA framework, which aims to enable proactive self-adaptation. To this end, PROSA exploits online testing techniques to detect changes and deviations before they can lead to undesired consequences. This paper introduces and illustrates the key online testing activities needed to trigger proactive adaptation, and it discusses how those activities can be implemented by utilizing and extending existing testing and adaptation techniques.

1 Introduction

Service-based applications operate in highly dynamic and flexible contexts of continuously changing business relationships. The speed of adaptations is a key concern in such a dynamic context and thus there is no time for manual adaptations, which can be tedious and slow. Therefore, service-based applications need to be able to self-adapt in order to timely respond to changes in their context or their constituent services, as well as to compensate for deviations in functionality or quality of service. Such adaptations, for example, include changing the workflow (business process), the service composition or the service bindings.

In current implementations of service-based applications, monitoring events trigger the adaptation of an application. Yet, monitoring only observes changes or deviations *after* they have occurred. Such a reactive adaptation has several important drawbacks. First, executing faulty services or process fragments may have undesirable consequences, such as loss of money and unsatisfied users. Second, the execution of adaptation activities on the running application instances can considerably increase execution time, and therefore reduce the overall performance of the running application. Third,

* The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

it might take some time before problems in the service-based application lead to monitoring events that ultimately trigger the required adaptation. Thus, in some cases, the events might arrive so late that an adaptation of the application is not possible anymore, e.g., because the application has already terminated in an inconsistent state.

Proactive adaptation presents a solution to address these drawbacks, because – ideally – the system will detect the need for adaptation and will self-adapt before a deviation will occur during the actual operation of the service-based application and before such a deviation can lead to the above problems.

In this paper we introduce the *PROSA* framework for *PRO*-active *S*elf-*A*daptation. PROSA's novel contribution is to exploit online testing solutions to proactively trigger adaptations. Online testing means that testing activities are performed during the operation phase of service-based applications (in contrast to offline testing which is done during the design phase). Obviously, an online test can fail; e.g., because a faulty service instance has been invoked during the test. This points to a potential problem that the service-based application might face in the future of its operation; e.g., when the application invokes the faulty service instance. In such a case, PROSA will proactively trigger an adaptation to prevent undesired consequences.

The remainder of the paper is structured as follows: In Section 2 we give an overview of current research results on using monitoring to enable (reactive) adaptation and of the state-of-the-art in online and regression testing. In Section 3 we present the PROSA framework. While describing the key elements of the framework, we discuss how those could be implemented by utilizing or extending existing testing and adaptation techniques. Section 4 introduces several application scenarios to illustrate how PROSA addresses different kinds of deviations and changes. Finally, Section 5 critically reviews the framework and highlights future research issues.

2 State-of-the-Art

2.1 Monitoring for Adaptation

Existing approaches for adaptation of service-based applications rely on the possibility to identify and realize – at run-time – the necessity to change certain characteristics of an application. In order to achieve this, adaptation requests are explicitly associated to the relevant events and situations. *Adaptation requests* (also known as adaptation requirements or specifications) specify how the underlying application should be modified upon the occurrence of the associated event or situation. These events and situations may correspond to various kinds of failures (like application-level exceptions and infrastructure-level failures), changes in contextual settings (like execution environment and usage context), changes among available services and their characteristics, as well as variations of business-level properties (such as key performance indicators).

In order to detect these events and situations, the majority of adaptation approaches resorts to exploiting *monitoring* techniques and facilities, as they provide a way to collect and report relevant information about the execution and evolution of the application. Depending on the goal of a particular adaptation approach, different kinds of events are monitored and different techniques are used for this purpose.

In many approaches (e.g., [1–4]) the events that trigger the adaptation are failures. These failures include typical problems such as application exceptions, network problems and service unavailability [1, 4], as well as the violation of expected properties and requirements. In the former case fault monitoring is provided by the underlying platform, while in the latter case specific facilities and tools are necessary. In [2] Baresi et al. define the expected properties in the form of WS-CoL assertions (pre-, post-conditions, invariants), which define constraints on the functional and quality of service (QoS) parameters of the composed process and its context. In [5] Spanoudakis et al. use properties in the form of complex behavioral requirements expressed in event calculus. In [3] Erradi at al. express expected properties as policies on the QoS parameters in the form of event-condition-action (ECA) rules. When a deviation from the expected QoS parameters is detected, the adaptation is initiated and the application is modified. In such a case, adaptation actions may include re-execution of a particular activity or a fragment of a composition, binding/replacement of a service, applying an alternative process, as well as re-discovering and re-composing services. In [6] Siljee et al. use monitoring to track and collect the information regarding a set of predefined QoS parameters (response time, failure rates, availability) infrastructure characteristics (load, bandwidth) and even context. The collected information is checked against expected values defined as functions of the above parameters, and in case of a deviation, the reconfiguration of the application is triggered.

Summarizing, all these works follow the reactive approach to adaptation, i.e., the modification of the application takes place *after* the critical event happened or a problem occurred.

The situation with reactive adaptation is even more critical for approaches that rely on post-mortem analysis of the application execution. A typical monitoring tool used in such approaches is the analysis of process logs [7–9]. Using the information about histories of application executions, it is possible to identify problems and non-optimalities of the current business process model and to find ways for improvement by adapting the service-based application. However, once this adaptation happens, many process instances might have already been executed in a "wrong" mode.

2.2 Online Testing and Regression Testing

The goal of testing is to systematically execute service instances or service-based applications (service compositions) in order to uncover failures, i.e., deviations of the actual functionality or quality of service from the expected one.

Existing approaches for testing service-based applications mostly focus on testing during design time, which is similar to testing of traditional software systems. There are a few approaches that point to the importance of online testing of service-based applications. In [10] Wang et al. stress the importance of online testing of web-based applications. The authors, furthermore, see monitoring information as a basis for online testing. Deussen et al. propose an online validation platform with an online testing component [11] . In [12] metamorphic online testing is proposed by Chan et al., which uses oracles created during offline testing for online testing. Bai et al. propose adaptive testing in [13, 14], where tests are executed during the operation of the service-based application and can be adapted to changes of the application's environment or of the application itself. Finally, the role of monitoring and testing for validating service-based applications is examined in [15], where the authors propose to use both strategies in combination. However, all these approaches do not exploit testing results for (self-)adaptation.

An approach related to online testing is regression testing. Regression testing aims at checking whether changes of (parts of) a system negatively affect the existing functionality of that system. The typical process is to re-run previously executed test cases. Ruth et al. [16, 17] as well as Di Penta et al. [18] propose regression test techniques for Web services. However, none of the techniques addresses how to use test results for the adaptation of service-based applications.

Summarizing, in spite of a number of approaches for online testing and regression testing, none of these approaches targets the problem of proactive adaptation. Still, several of the presented approaches provide baseline solutions that can be utilized and extended to realize online testing for proactive adaptation. This will be discussed in the following section.

3 PROSA: Online Testing for Proactive Self-Adaptation

As introduced in Section 1, the novel contribution of the PROSA framework is to exploit online testing for proactive adaptation. Therefore, the PROSA framework prescribes the required online testing activities and how they lead to adaptation requests. Figure 1 provides an overview of the PROSA framework and how the proactive adaptation enabled by PROSA relates to "traditional" reactive adaptation which is enabled by monitoring.

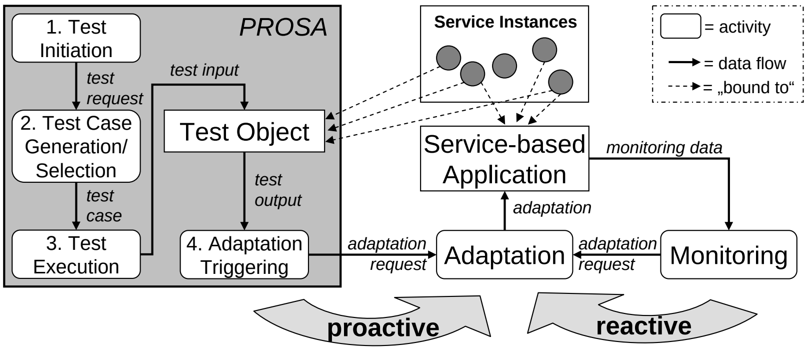


Fig. 1. The PROSA Framework

The PROSA framework prescribes the following four major activities:

1. *Test initiation*: The first activity in PROSA is to determine the need to initiate online tests during the operation of the service-based application. The decision on when to initiate the online tests depends on what kind of change or deviation should be uncovered (see Section 3.1).

2. *Test case generation/selection*: Once online testing has been initiated by activity 1, this second activity determines the test cases to be executed during online testing. This can require creating new test cases or selecting from already existing ones (see Section 3.2).
3. *Test execution*: The test cases from activity 2 are executed (see Section 3.3).
4. *Adaptation triggering*: Finally, an analysis of the test results provides information on whether to adapt the service-based application and thus to create adaptation requests (see Section 3.4).

It should be noted that – as depicted in Figure 1 – online testing does not interfere with the execution of the actual application in operation, i.e. with those instances of the application which are currently used by actual users. Rather, online testing performs tests of the constituent parts of the service-based application (e.g., individual services or service compositions) independent from and in parallel to the operating applications.

Details about the above activities and how those can be implemented with existing techniques are discussed in the remainder of this section.

3.1 Test initiation

In order to initiate the actual online testing activities (PROSA's activities 2 and 3), two questions need to be answered: "When to test?" and "What to test?". The answer of these questions depends on the kinds of changes or deviations that should be *proactively* addressed in addition to reactive techniques like monitoring. Those possible kinds of changes are listed in Table 1.

Table 1. Different cases for initiating online testing

Case	*Why to initiate online testing?*	*When to initiate online testing?*	*What to test?*
1	Uncovering failures introduced due to the adaptation of the service-based application.	Once the respective adaptation (e.g. binding of a new service) has been performed.	service or composition
2	Detecting changes in the service-based application or its context that could lead to failures in the "future".	Once monitoring has detected a change that does not reactively trigger an adaptation.	service or composition
3	Identifying failures of an application execution.	Periodically (e.g. randomly or by testing future service invocations along the execution path of the application).	composition
4	Uncovering failures (i.e., deviations from expected functionality or quality) or unavailability of constituent services.	Periodically (e.g., randomly or by predicting future service invocations along the execution path of the application).	service

To give an answer to the question "When to test?", Table 1 provides an explanation when to initiate online testing depending on the kind of change or deviation. Those kinds of changes and deviations are illustrated in more detail in Section 4, where different application scenarios for PROSA are introduced.

To provide an answer to the question "What to test?" (i.e., to determine the test object), we have considered the following two major strategies that can be performed in order to uncover the different kinds of changes or deviations (Table 1 shows what strategy could be followed depending on the kind of change or deviation):

- *Testing constituent service instances:* Similar to unit or module testing, the individual, constituent service instances of a service-based application can be tested (i.e., the service instances that are or will be bound to the service-based application).
- *Testing service compositions:* Similar to system and integration testing, the complete service composition of a service-based application or parts thereof can be tested.

To implement activity 1 of PROSA, one can rely on information provided by existing monitoring techniques for case 2 (see Table 1) or adaptation techniques for case 1. The other cases require new and specific techniques, which can be very simple (like randomly triggering the tests) or more challenging (like predicting future service invocations along the execution path of the application).

3.2 Test case generation/selection

In Section 3.1 two strategies for online testing were introduced. In order to implement these two different strategies and thus to realize activity 2 of the PROSA framework, different kinds of techniques for determining test cases have to be employed:

- *Testing constituent service instances:* For testing constituent service instances, existing techniques for test case generation from service descriptions, like WSDL, can be exploited (e.g., [19–21]). Additionally, test cases from the design phase can be re-used if such test cases exist. However, usually the test cases from the design phase will not suffice, because typically at that time not all services are known due to the adaptation of a service-based application that can happen during run-time.
- *Testing service compositions:* For testing service compositions, test cases can be generated from composition specifications, like BPEL (e.g., [22, 23]). If a set of test cases for testing service compositions already exists, online testing has to determine which of those test cases to execute again (i.e., test cases have to be selected). This is similar to regression testing, which has been discussed in Section 2.2. Consequently, existing techniques for regression testing of services (like [16–18]) can be utilized.

A more detailed survey on existing test case generation and selection techniques for service-based applications can be found in [24].

3.3 Test execution

The responsibility of activity 3 in the PROSA framework is to execute the test cases that have been determined by activity 2. This means that the test object (which is either a service instance or a service composition) is fed with concrete inputs (as defined in the test cases) and the produced outputs are observed.

The test execution can be implemented by resorting to existing test execution environments, e.g., the ones presented in [19, 18]. It is important to note that invoking services can lead to certain "side effects" which should not occur when invoking the service for testing purposes only (this problem is also discussed in [22]). As an example, when invoking the service of an online book seller for testing purposes, one would not like to have the "ordered" books actually delivered. Thus, it is necessary to provide certain services with a dedicated test mode. As an example, one could follow the approaches suggested for testing software components, where components are provided with interfaces that allow the execution of the component in "normal mode" or in "test mode" (see [25]).

3.4 Adaptation triggering

The final activity 4 of PROSA determines whether to issue an adaptation request, which ultimately leads to the modification of the service-based application. Such an adaptation request should be issued when the observed output of a test deviates from the expected output, i.e., whenever a test case fails. This includes deviations from the expected functionality as well as from the expected quality of service.

As has been discussed above, existing adaptation solutions rely on monitoring to issue adaptation requests whenever a deviation is observed (see reactive loop in Figure 1). In order to exploit those existing solutions (see Section 2.1), triggering of adaptations based on online testing should conform to the requests from the monitoring component. Thereby, activity 4 could be implemented within a unified adaptation framework.

To achieve such a unification, the following two issues need to be resolved: First, specific adaptation requests should be explicitly assigned to individual test cases. In reactive approaches such adaptation requests are assigned to certain monitoring events. The events may represent application or network failures (e.g., service is unavailable), violation of assertions (e.g., post-condition on data returned by service call) or even of complex behavioral properties (e.g., if flight is found but there are no rooms available, the trip plan can not be created). In a similar way, test cases represent dedicated execution scenarios, where specific deviations or changes can be checked (this has been highlighted in Table 1). If the test fails, this is similar to the occurrence of a monitoring event, and thus the adaptation assigned to the test case is triggered.

Second, it may be necessary to modify the adaptation requests from monitoring in order to take into account the specifics of proactive adaptation. Indeed, some adaptation requests from monitoring might specify instructions that are not applicable in proactive adaptation (e.g., "retry" operation, or "rollback to safe point"). Therefore, the specification should be changed such that these instructions do not appear when used for proactive adaptation. An interesting line of future work in these regards could be to devise means to automatically derive adaptation requests for proactive adaptation from the adaptation requests already available for monitoring.

4 Application Scenarios

In this section we illustrate how PROSA enables the proactive adaptation of a service-based application. For this purpose we introduce an example application based on which we describe scenarios that demonstrate how PROSA can be applied to the different cases for online testing introduced in Table 1. The service composition of the example and possible constituent service instances are depicted in Figure 2.

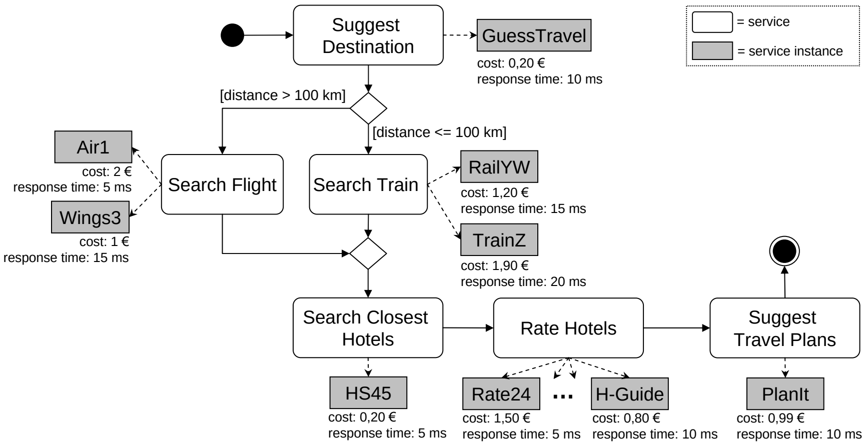


Fig. 2. Example Application: "Travel Planning"

Our example application provides a travel planning service, which includes a combined search for transportation and hotel accommodation. The constituent services of this application are invoked in the following order:

1. *Suggest destination:* First, the user of the application is provided with a suggestion of different travel destinations based on her/his preferences.
2. *Search flight/train:* Once the user has chosen a destination, the application will determine the best way to reach that destination. Depending on the distance to the suggested destination, either an appropriate flight or a train connection is searched.
3. *Search closest hotels:* After a suitable means of transportation has been found, hotels in the vicinity of the airport or the railway station of the destination are located.
4. *Rate hotels:* Using one of the many hotel rating services available, each hotel from the list is checked for its rating and the hotel list, sorted according to the rating, is returned.
5. *Suggest travel plans:* Finally, the first hotel from the sorted list (i.e., the one with the best overall rating) is chosen and the travel information (itineraries, information about the hotel, etc.) is compiled to produce a comprehensive travel plan.

In Figure 2, gray boxes denote concrete service instances that can be bound to the application in order to compute the travel plan. Some of those concrete service instances can already be known at design time, while others are dynamically discovered or added due to adaptations during the operation of the service-based application. The annotated information about cost and response time denotes the negotiated quality for each of the service instances (e.g., by means of service level agreements).

4.1 Case 1: Failure introduced due to adaptation

Let us assume that the service instance “H-Guide” was bound to our service-based application at operation time, because the service instance “Rate24” has turned out to be too expensive. The binding of that new service instance is reported by the adaptation component to the PROSA framework. Consequently, PROSA’s activity 1 triggers the online testing activities, which react to this adaptation by determining test cases to check whether the newly bound service instance behaves as expected (see Table 1, case 1). Let us say that the expected output of one of those test cases is “Palermo Premium Class Hotel”, which clearly is the hotel with the best ratings for the chosen location. Unfortunately, the observed output of “H-Guide” is “Casa Palermo”, which is the hotel with one of the lowest ratings (the reason for this presumed failure is that – other than expected – “H-Guide” returns the list of hotels in ascending order, starting with the lowest ratings). Online testing reports this failure to the adaptation component, which can – for example – switch back to the initial service instance “Rate24”, which has already been used successfully.

4.2 Case 2: Change that could lead to failures in the future

Let us assume that a new regulation concerning the pricing of flights enters into force during the operation of the service-based application. The regulation requires that the overall cost of a flight (including taxes) has to be stated and that it may not anymore be stated as the price for the flight with the note “plus taxes”. This legal change thus represents a change in the context of the application (see Table 1, case 2). As a result, PROSA will initiate online testing activities – when this new regulation enters into force – in order to determine whether the constituent service instances of the service-based application conform to this new regulation. This means that online tests will be triggered in order to check whether the service instances for flight booking (“Air1” and “Wings3”) conform to the new regulation. If one of those service instances does not implement the new regulation, the service-based application will be adapted accordingly before that service instance is invoked during the actual operation of the application.

4.3 Case 3: Failure of an application execution

The output of “search train” (resp.“search flight”) contains the name of the city close to the airport or the railway station. This city name is passed on to “search closest hotels” in order to determine the list of hotels in the vicinity of the destination. Let us assume that the service instance “RailYW” always provides the name of the destination in “short” form, meaning that even if there is more than one city with this name, like “Frankfurt am Main” and “Frankfurt an der Oder”, this service instance will always return “Frankfurt”. When the hotel searching service “HS24” receives such an ambiguous input, it will terminate with an error message. By running test cases to check deviations in the service composition (see Table 1, case 3), PROSA can uncover such a failure and – as a proactive corrective action – can request that a different service instance is bound to the application (e.g., "TrainZ").

4.4 Cases 4: Failure of a constituent service

For the booking of an appropriate flight, two service instances are available: "Air1" and "Wings3". "Air1" is used for premium clients, which are willing to pay more for a shorter response time. "Wings3" is the preferred choice of clients who want to save money. At operation time the online testing component runs several test cases per hour (periodically testing, see Table 1, case 4). Let us assume that one of those tests uncovers that "Wings3" does not respond. PROSA then provides the adaptation component with this information, such that the alternative service instance "Air1" (which is working as expected) is used for all queries.

5 Discussion and Perspectives

This paper has introduced the PROSA framework, which defines key activities for enabling the proactive self-adaptation of service-based applications. The novel contribution of PROSA is to exploit online testing techniques in order to anticipate future deviations or changes of a service-based application and thereby to trigger adaptation requests. In addition to the definition of those key activities, the paper has discussed how those activities can be implemented by building on or extending existing testing and adaptation techniques.

In contrast to the "traditional" form of reactive adaptation (e.g., based on monitoring), PROSA provides the following important benefits: First, changes or deviations from expected functionality or quality of service can be uncovered and addressed before they lead to undesirable consequences. Second, the execution of adaptation activities – if done proactively – does not interfere with the execution of the actual application instances, i.e., the users of the application won't be affected by the adaptation. Third, proactive adaptation can provide adaptation requests early enough such that an adaptation of the service-based application still is possible (in contrast to reactive adaptation, where the application can have already terminated in an inconsistent state, for instance). Due to these benefits, we are confident that the PROSA framework will enable novel service-based applications that are able to proactively adapt and thus to better meet their expectations.

In addition to uncovering failures, monitoring is also often used to improve (or optimize) a service-based application. Accordingly, online testing could be used in this respect, for instance by determining the best possible alternative for an adaptation decision before the adaptation is executed. This means whenever an adaptation decision is imminent and different alternatives exist, those alternatives could be "pre-tested" and the best one chosen. For example, consider an adaptation specification, where on failure of a service instance three strategies are defined: retry invoking the service instance three times, replace the service instance with another service instance, change the service composition to use different services. Testing can now "simulate" all those three strategies and maybe detect that "change composition" is the only way to successfully drive the adaptation.

Although exploiting only testing for proactive adaptation provides many benefits, we acknowledge at this stage that further work is required in order to demonstrate the applicability of the PROSA idea in practice. One aspect that, for example, has to be investigated, is the possible impact of the execution of test cases on the performance of the application. Thus, key issues that we will target in our future work are to create a proof-of-concept prototypes based on existing techniques and tools (as discussed in the paper) and to apply these prototypes to realistic cases.

As we have briefly pointed out in the paper, proactive and reactive adaptation may work together in an integrated dynamic adaptation framework. In such a framework, online testing and monitoring could mutually benefit from each other, thereby improving the overall quality and efficiency of adaptation. In further work, we thus plan to investigate on how to best exploit the synergies between monitoring and testing. As an example, the results of monitoring may be used to identify "better" test cases for online testing. When complex behavioral properties are monitored (e.g., see [26, 5]), the violations or successful executions are represented as traces containing information of the composition activities. A set of such traces from previous executions may be used to derive new test cases for online testing. Furthermore, monitoring may be used to parametrize the test cases. As the configuration of tests may depend on the operational context of the application, such context information can be provided by monitoring.

References

1. Baresi, L., Ghezzi, C., Guinea, S.: Towards Self-healing Service Compositions. In: First Conference on the PRInciples of Software Engineering (PRISE'04). (2004) 11–20
2. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: ESSPE '07: International workshop on Engineering of software services for pervasive environments. (2007) 11–20
3. Erradi, A., Maheshwari, P., Tosic, V.: Policy-Driven Middleware for Self-adaptation of Web Services Compositions. In: ACM/IFIP/USENIX 7th International Middleware Conference. (2006) 62–80
4. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In: 1st workshop on Middleware for Service Oriented Computing. (2006) 48–53
5. Spanoudakis, G., Zisman, A., Kozlenkov, A.: A Service Discovery Framework for Service Centric Systems. In: SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing. (2005) 251–259
6. Siljee, J., Bosloper, I., Nijhuis, J., Hammer, D.: DySOA: Making Service Systems Self-adaptive. In: 3rd International Conference Service-Oriented Computing - ICSOC 2005. (2005) 255–268
7. van der Aalst, W.M.P., Pesic, M.: Specifying and Monitoring Service Flows: Making Web Services Process-Aware. In Baresi, L., Di Nitto, E., eds.: Test and Analysis of Web Services. Springer (2007) 11–55
8. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining - Adaptive Process Simplification Based on Multi-perspective Metrics. In: Business Process Management, 5th International Conference, BPM. (2007) 328–343

9. Nezhad, H.R.M., Saint-Paul, R., Benatallah, B., Casati, F.: Deriving Protocol Models from Imperfect Service Conversation Logs. IEEE Transactions on Knowledge and Data Engineering (TKDE) (2008) to appear.
10. Wang, Q., Quan, L., Ying, F.: Online testing of Web-based applications. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC). (2004) 166–169
11. Deussen, P., Din, G., Schieferdecker, I.: A TTCN-3 based online test and validation platform for Internet services. In: Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS). (2003) 177–184
12. Chan, W., Cheung, S., Leung, K.: A metamorphic testing approach for online testing of service-oriented software applications. International Journal of Web Services Research **4** (2007) 61–81
13. Bai, X., Chen, Y., Shao, Z.: Adaptive web services testing. In: 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 233–236
14. Bai, X., Xu, D., Dai, G., Tsai, W., Chen, Y.: Dynamic reconfigurable testing of service-oriented architecture. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 368–375
15. Canfora, G., di Penta, M.: SOA: Testing and Self-checking. In: Proceedings of International Workshop on Web Services - Modeling and Testing - WS-MaTE. (2006) 3 – 12
16. Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., Tu, S.: Towards automatic regression test selection for web services. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 729–734
17. Ruth, M., Tu, S.: A safe regression test selection technique for Web services. In: Second International Conference on Internet and Web Applications and Services (ICIW). (2007)
18. Di Penta, M., Bruno, M., Esposito, G., et al.: Web Services Regression Testing. In Baresi, L., Di Nitto, E., eds.: Test and Analysis of Web Services. Springer (2007) 205 – 234
19. Martin, E., Basu, S., Xie, T.: Automated Testing and Response Analysis of Web Services. In: IEEE International Conference on Web Services (ICWS). (2007) 647 – 654
20. Bai, X., Dong, W., Tsai, W.T., Chen, Y.: WSDL-Based Automatic Test Case Generation for Web Services Testing. In: Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE), IEEE Computer Society (2005) 215 – 220
21. Tarhini, A., Fouchal, H., Mansour, N.: A simple approach for testing Web service based applications. In: 5th International Workshop on Innovative Internet Community Systems. Lecture Notes in Computer Science Vol.3908 (2006) 134–146
22. Lübke, D.: Unit Testing BPEL Compositions. In Baresi, L., Di Nitto, E., eds.: Test and Analysis of Web Services. Springer (2007) 149 – 171
23. Dong, W.L., Yu, H., Zhang, Y.B.: Testing BPEL-based Web Service Composition Using High-level Petri Nets. In: EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, IEEE Computer Society (2006) 441–444
24. Pernici, B., Metzger, A., eds.: Survey of quality related aspects relevant for SBAs. (2008) S-Cube project deliverable: PO-JRA-1.3.1. http://www.s-cube-network.eu/achievements-results/s-cube-deliverables.
25. Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT approach to runtime component testing. In: Proceedings of the 30th Annual Int'l Computer Software and Applications Conference (COMPSAC). (2006) 171–176
26. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of Instances and Classes of Web Service Compositions. In: IEEE International Conference on Web Services (ICWS 2006). (2006) 63–71

Towards Pro-active Adaptation with Confidence – Augmenting Service Monitoring with Online Testing

Andreas Metzger
Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen
Schützenbahn 70, 45127 Essen, Germany
andreas.metzger@sse.uni-due.de

Osama Sammodi
Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen
Schützenbahn 70, 45127 Essen, Germany
osama.sammodi@sse.uni-due.de

Klaus Pohl
Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen
Schützenbahn 70, 45127 Essen, Germany
klaus.pohl@sse.uni-due.de

Mark Rzepka
Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen
Schützenbahn 70, 45127 Essen, Germany
mark.rzepka@sse.uni-due.de

ABSTRACT

Service-based applications need to operate in a highly dynamic and distributed world. As those applications are composed of individual services, they have to react to failures of those services to ensure that the applications maintain their expected functionality and quality. Self-adaptation is one solution to this problem, as it allows applications to autonomously react to failures. Currently, monitoring is typically used to identify failures, thus triggering adaptation. However, monitoring only observes failures after they have occurred, which means that adaptation based on monitoring is reactive. This can lead to shortcomings like user dissatisfaction, increased execution times, and late response to critical events. Proactive adaptation addresses those shortcomings, because in such a setting, the application detects the need for adaptation and thus can adapt before a failure will occur. However, it is important to avoid unnecessary pro-active adaptations, as they can lead to severe shortcomings, such as increased costs or follow-up failures. This means that when taking pro-active adaptation decisions it is key that there is confidence in the predicted future failures, i.e., pro-active adaptation should only be performed if there is certainty that the failure could in fact occur. To avoid unnecessary adaptations, we introduce an approach based on augmenting service monitoring with online testing to produce failure predictions with confidence. We demonstrate the applicability of our approach using a scenario from the eGovernment domain.

1. MOTIVATION

Service-based applications (SBAs) need to operate in a highly dynamic and distributed world. Especially, as an SBA is composed of individual services, an SBA has to react to changes and failures of its constituent services to ensure that the SBA maintains its expected functionality and quality. In such a setting, monitoring is typically used to identify failures of the constituent services. To compensate for those failures, an adaptation of the SBA can be triggered [1, 2].

Monitoring only observes failures after they have occurred, and thereby only allows a *reactive* adaptation of the failed SBA instances. Such a reactive adaptation, however, has several important drawbacks [3]. Firstly, it might take some time before problems in an SBA instance lead to monitoring events that ultimately trigger the required adaptation. Thus, in some cases, the events might arrive so late that an adaptation of the SBA instance is not possible anymore, e.g., because the application instance has already terminated in an inconsistent state. Secondly, the execution of adaptation activities on the running application instances can considerably increase execution time, and therefore reduce the overall performance of the running application. Thirdly, executing faulty services may have undesirable consequences, such as loss of money and unsatisfied users.

In cases where more than one SBA instance is running, those drawbacks can be mitigated through *pro-active* adaptation. Assume that, as an example, a service S is used in several SBA instances. If S fails during the execution of one SBA instance, the other running SBA instances could be modified before they fail. This could be done, e.g., by replacing S with an alternative service S', provided that S has not yet been invoked in those SBA instances.

1.1 Problem Statement

To enable pro-active adaptation, future failures need to be predicted. However, when taking pro-active adaptation decisions it is key that there is confidence in the predicted future failures. This means that pro-active adaptation should only be performed if there is a high enough certainty (e.g., such as 95%) that the failure addressed by that adaptation could in fact occur. Especially, in the case where an SBA is built from external (third party) services and thus those constituent services are not under the control of the service composer, the observed quality and functionality of those constituent services can vary between different service invocations. As an example, a failure observed at one point in time can disappear at a later point in time, as a service provider could have repaired the service in the meantime. This means that even if a constituent service S fails during the execution of one SBA instance, it might well work as expected when invoked for other SBA instances. Thus, if such failures were used as a basis for adaptation decisions, this would lead to unnecessary adaptations of those SBA instances, i.e.,

SEAMS '10, May 2-8, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-971-8 ...$10.00.

even if those adpatations were not performed, the SBA instances would still work as expected.

Such unnecessary adaptations can have severe shortcomings: Firstly, unnecessary adaptations can be costly even in the pro-active case. For instance, additional activities such as Service Level Agreement (SLA) negotiation for the alternative services might have to be performed, or the adaptation can lead to a more costly operation of the SBA, e.g., if a seemingly unreliable but cheap service is replaced by a more costly one. Secondly, unnecessary adaptations could be faulty (e.g., if the new service has bugs), leading to severe problems as a consequence. Thus, unnecessary adaptations should be avoided as best as possible.

Currently, there exist several approaches to predict the failures of software and service systems (see Section 3). Many of them consider statistical techniques during monitoring (e.g., log correlation) or during testing (e.g., statistical testing). However, those approaches face a significant shortcoming when applied to adaptive SBAs: Those techniques usually require a very high number of data points (monitoring or test results) to produce statistically significant data, i.e., if not enough data points are available, the expected statistical significance and thus the expected confidence in the failure prediction might not be reached. This poses significant problems for the applicability of those techniques in the SBA context. If monitoring approaches are applied and only few users have started to use the SBA, the collected monitoring data will usually not suffice to predict future failures with confidence. If testing approaches are applied individually, achieving the required number of data points can lead to significant, additional costs. This is especially true if the SBA includes external services, and thus invoking those services will be associated with additional costs.

1.2 Contribution of Paper

In this paper, we take the position that monitoring of an SBA should be augmented with online testing in order to produce failure predictions with confidence, and thus, avoid unnecessary adaptation. With online testing we mean that the SBA is tested (i.e, fed with dedicated test input) in parallel to its normal use and operation. The online test cases are determined in such a way, that the results provide additional data points that complement the data points collected by monitoring.

As we build on the monitoring data available, we expect that due to the synergy effects between monitoring and testing, the testing effort can be kept smaller than if the techniques were applied individually. Although the integration of monitoring and testing has been proposed in the literature (e.g., to exploit test cases for monitoring; see Section 3.4), the novel way of integrating monitoring and testing as introduced in this paper has not been proposed so far.

The remainder of the paper is structured as follows: Section 2 introduces an example scenario based on which the problem addressed in this paper is illustrated. Although our approach is general in nature, for illustration purposes we focus on service response time as a quality attribute and the replacement of service as a healing technique. Section 3 discusses the shortcomings of the state of the art and how we envision to progress from that. Section 4 outlines our overall approach together with its assumptions. Section 5 illustrates the applicability of our approach by using the example scenario from Section 2 and it discusses how the assumptions underlying our approach could be relaxed. Section 6 concludes the paper and provides an outlook on future work.

2. ILLUSTRATION

2.1 Example Scenario

In this section, we illustrate the problem using a scenario that is based on an eGovernment case study as described in [4]. The scenario specifies the usage of a governmental SBA that allows citizens to renew their vehicle registrations online, thereby sparing them effort and time.

The service composition and workflow of the scenario are depicted in Figure 1 as an activity diagram. Gray boxes denote concrete services that can be bound to the eGovernment application. A gray box with a dotted border represents an internal service. A gray box with a solid border represents a third-party service (i.e., a service provided by an external organization). In addition, the diagram is annotated with information about the negotiated response times (e.g., stipulated by means of SLAs for third-party services) and the costs involved in service invocations.

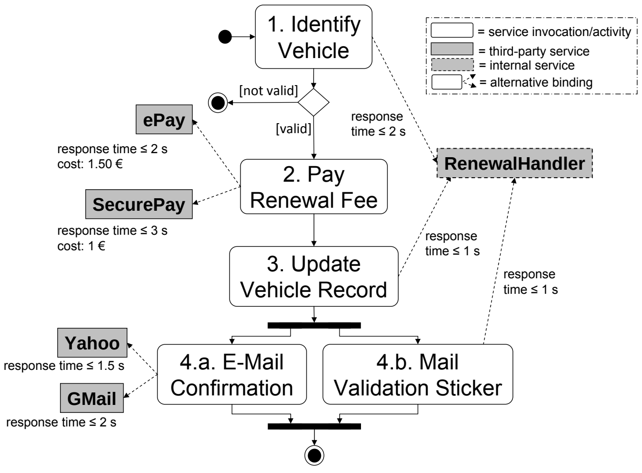


Figure 1: Workflow of eGovernment Application.

The workflow consists of the following activities:

1. *Identify Vehicle:* In order to begin online registration renewal, the citizen needs to provide a renewal identification number from the vehicle registration notice or, as an alternative, she/he could provide the license plate number. This information is needed to identify and to find the matching vehicle record.

2. *Pay Renewal Fee:* Once the vehicle – for which the registration renewal will be made – has been identified, the citizen will be asked to pay the renewal registration fee. For this purpose, the citizen will have to interact with a payment service and await the confirmation of the payment.

3. *Update Vehicle Record:* After the payment of the fee is completed, the application will renew the registration of the vehicle and will update the vehicle record to reflect the registration renewal.

4. *E-Mail Confirmation and Mail Validation Sticker:* Finally, a confirmation of the online transaction (i.e., the registration renewal process) will be sent to the citizen by e-mail. In parallel to that, the motor vehicle division (i.e., the devision in charge of vehicle registration) will be notified about the registration renewal and will mail a validation sticker to the citizen.

2.2 Pro-active Adaptation

To illustrate the benefits of pro-active adaptation, let us assume that the *SecurePay* service fails, i.e., takes longer than the allowed maximum response time of 3 seconds, when a citizen tries to pay the renewal fee using the eGovernment application. Let us also assume that there are other instances of the eGovernment application that are running in parallel but have not yet invoked *SecurePay* (we call them yet-to-invoke-SecurePay instances). Those instances could be prevented from failing by exploiting pro-active adaptation. In our example, the *SecurePay* service could be replaced by the *ePay* service in the yet-to-invoke-SecurePay instances.

Based on the previous situation, we can now illustrate the problem of unnecessary pro-active adaptation. Let us assume that the yet-to-invoke-SecurePay instances were pro-actively adapted but this adaptation turns out to be unnecessary. In our example, the adaptation was triggered by the failure of the *SecurePay* service, which could turn out to work well when being invoked at a later point in time, i.e., the service would have worked as expected for the yet-to-invoke-SecurePay instances (i.e., it would have responded within 3 seconds). Unfortunately, this unnecessary pro-active adaptation has the following consequences in our example. Firstly, the adaptation has lead to increased costs due to the replacement of the cheaper *SecurePay* service (1 €) with the more costly *ePay* service (1.50 €). Secondly, the situation gets even worse if the *ePay* service turns out to be faulty. In such a case the unnecessary pro-active adaptation has replaced the *SecurePay* service that was working well with the faulty service *ePay*, and this replacement would eventually make the yet-to-invoke-SecurePay instances fail.

3. RELATED WORK

Various techniques to uncover and predict failures have been proposed in the literature on assuring the quality software systems and SBAs. This section first provides basic definitions of the different classes of quality assurance techniques and discusses their respective benefits and shortcomings. Then, for each of those classes, this section will discuss the most relevant techniques for addressing the problem introduced in Section 1. To this end, first monitoring, testing and (static) analysis are discussed individually, followed by a discussion of joint efforts for monitoring and testing.

3.1 Monitoring

Monitoring in General: Monitoring *observes* services or service-based applications during their current execution, i.e. during their actual use or operation (cf., [1, 2]) with the aims of, for instance, supporting the optimization, enabling the context-driven adaptation, or uncovering failures. Many different monitoring techniques have been presented in the literature (see [2]).

In contrast to testing and static analysis (see below), which aim at providing more general statements about services or service-based applications, monitoring provides statements about a service-based application's current execution (i.e., about current execution traces). Thereby, monitoring can uncover failures which have escaped testing, because the concrete input that lead to the current execution trace might have not been covered by any test case.[1]

Data correlation: In the literature several approaches have been proposed to statistically analyze failure data (retrieved from event logs or monitoring logs) in order to develop prediction models that are used for predicting failure patterns. One prominent class of approaches is to examine the spatial and/or temporal correlation among failure events to determine such patterns. Depending on the types of failures, failure events may display different relationships: (1) Spatial correlations (e.g., a failure may nearly simultaneously occur on multiple nodes in a cluster) as explored in [5, 6, 7, 8, 9], and (2) temporal correlations (e.g., some faults cause several failure instances occurred on multiple compute nodes in a short interval), as explored in [2, 4, 5]. These temporal and spatial correlation properties of failure events have been utilized for failure prediction and proactive management in several of those approaches [2, 3, 4].

Although those approaches address the problem of failure prediction, our envisioned approach differs from them in at least two important aspects. Firstly, we don't aim at predicting new failures from past monitoring data, but want to establish *confidence* that one specific failure which has been uncovered in one SBA instance could also occur in other SBA instances and thus justifies a pro-active adaptation of those other SBA instances. Secondly, the developed failure prediction models typically are built from event logs that are collected over a very large period of time (e.g., more than 100 days in [2], and a year in [1]), whereas in the SBA context we cater for dynamic adaptations of the systems and thus need to consider a possible invalidation of past monitoring data due to those adaptations.

A related approach is presented in [10] for anomaly detection and localization based on statistical machine learning techniques. To this end, the authors propose building and periodically updating one or more baseline system models by observing the system's behavior. The models may capture either operational (time series) or structural behaviors. This approach aims at reactive adaptation, whereas our approach is targeted towards pro-active adaptation.

3.2 Testing

Testing in General: The goal of testing is to systematically *execute* a service or a service-based application (the test object) in order to uncover failures (cf. [11, 12, 13, 14]). During testing, the test object is fed with concrete inputs and the produced outputs are observed. The observed outputs can deviate from the expected outputs with respect to functionality as well as quality (e.g., performance or availability). When the observed outputs deviate from the expected outputs, a *failure* of the service or the service-based application is uncovered. Several testing techniques for services and service-based applications have been presented in the literature (see [15, 16]).

Testing cannot guarantee the absence of faults, because it is infeasible (except for trivial cases) to test all potential concrete inputs of the test object. As a consequence, a sub-set of all potential inputs has to be determined for testing (e.g., cf. [13]). The quality of the tests strongly depends on how well this sub-set covers the test object. Ideally this sub-set should include concrete inputs that are representative for all potential inputs (even those which are not tested) and it should include inputs that – with high probability – uncover failures. However, as choosing such an ideal sub-set typically is infeasible, it is important to employ other quality assurance techniques and methods which complement testing (e.g., cf. [14])

Statistical Testing: One important technique, which addresses the problem of determining feasible sub-sets of test cases while still maintaining adequate test coverage is statistical testing [17, 18]. The purpose of statistical testing is to predict the reliability of the test object. To this end, usage models are built, representing the system states and transitions between those states, together with probabilities for those state transitions. Those models are commonly represented in the form of Markov chains [19]. Usage models can be built from specifications, user guides, or by observing the user interactions with existing systems [17]. Test cases are generated from the usage models taking into account the transition

[1] As will be explained below, only a sub-set of all potential inputs can be tested.

probabilities. The results of the tests are statistically analyzed to determine the expected reliability of the system. Statistical testing has been used, for example, to test embedded systems (where usage models are derived from sequence-based requirements specification [20]), and to measure the reliability of Web applications (where usage models are derived from Web usage logs [21]).

Although statistical testing can provide statements about the overall system reliability (i.e., the probability that the system won't fail), it is not used for establishing confidence in predicting that a specific failure, which has been uncovered in one SBA instance, could also occur in other SBA instances. However, this more specific statement is needed in order to justify and enact pro-active adaptation of those SBA instances. Furthermore, statistical testing usually requires a very high number of test cases to produce statistically sound data and this poses a significant burden for the applicability of this technique in the SBA context. It is exactly here, where in our envisioned approach we propose to exploit failure data observed through monitoring. Finally, due to the dynamic and adaptive nature of SBAs, some of the assumptions that underlie statistical testing like that the "test object does not change" and that there are "fixed environmental conditions" [18] typically do not hold for SBAs.

Online Testing: Given the need for adapting SBAs at run-time, quality assurance techniques that can be applied at run-time are essential. The major type of run-time quality assurance techniques used today is monitoring (see Section 3.1). As monitoring observes the SBA (or its constituent services) during their current execution (i.e., during their actual use or operation), monitoring does not provide a systematic coverage of the 'test object'. First research activities have appeared that suggest bringing standard and consolidated software quality assurance techniques to run-time. In [22] the general need for run-time quality assurance techniques is motivated, focusing on automated techniques.

One promising run-time quality assurance technique is online testing. Online testing means that the service-based application is tested (by feeding it with dedicated test input) in parallel to its normal use and operation. In [23], Wang et al. stress the importance of online testing of web-based applications. The authors, furthermore, see monitoring information as a basis for online testing. Deussen et al. propose an online validation platform with an online testing component [24]. In [25], metamorphic online testing is proposed by Chan et al., which uses oracles created during offline testing for online testing. Bai et al. propose adaptive testing in [26, 27], where tests are executed during the operation of the service-based application and can be adapted to changes of the application's environment or of the application itself.

An approach related to online testing is regression testing. Regression testing aims at checking whether changes of (parts of) a system negatively affect the existing functionality of that system. The typical process is to re-run previously executed test cases. Ruth et al. [28, 29] as well as Di Penta et al. [30] propose regression test techniques for Web services.

Despite the fact that there are a number of initial approaches for online testing and regression testing, none of these approaches targets the problem of predicting the future occurrence of a failure with confidence, thereby avoiding unnecessary, proactive adaptations.

3.3 Static Analysis

Static Analysis in General: The aim of static analysis (e.g., see [31, 14]) is to systematically *examine* an artifact in order to determine certain properties or to ascertain whether some predefined properties are met. Examples of static analysis include formal techniques and methods, such as data flow analysis, model checking, and symbolic execution, as well as non-formal approaches, such as reviews, walk-throughs, and inspections.

In contrast to testing and monitoring, where individual executions of the services or service-based applications are examined, static analysis can examine classes of executions [14]. Thus, static analysis can lead to more universal statements about the properties of the artifacts than testing (or monitoring).

Static analysis is often based on a model of the system. As those models might abstract away from some relevant concrete details, aspects might be overlooked during static analysis [14] or simply not be captured faithfully enough. Thus static analysis can complement the other classes of quality assurance techniques and methods but typically will not be enough, if used in isolation, in order to give a complete picture of the execution of a computational system.

Reliability modeling and analysis: Software reliability is one of the most important characteristics of software quality. Software reliability is defined as the ability of a system to perform its required functions under stated conditions for a specified period of time. It is often reported in terms of a probability [32]. Software reliability techniques aim at reducing or eliminating failures in software systems. Reliability modeling is a major technique for the estimation as well as the prediction of the reliability of a software system.

Many reliability models have been proposed in the literature and could be classified as black-box models or white-box models. Black-box models consider the software system as a whole (monolithic entity) and ignore the internal structure of the system. Only interactions with the outside world are modeled [33]. White-box approaches on the other hand, consider a system's internal structure (e.g., architecture, states and execution paths) in reliability estimation [34, 35, 36, 37]. For SBAs it is important to reason on evolving and adapting the SBAs. To this end, Epifani et al. in [38] propose a framework for updating reliability models during the run-time of the SBA. They use a Bayesian estimator to produce updated model parameters from data collected from the running system.

Reliability analysis approaches are able to predict the overall reliability of the system (this is a similar goal to statistical testing, see Section 3.2). However, those techniques are not intended for predicting the occurrence of individual failures.

3.4 Joint Monitoring and Testing Efforts

As has been discussed above, isolated monitoring and testing techniques have certain shortcomings, which could be overcome by joint monitoring and testing efforts. For instance, Delgado et al. argue that monitoring provides a complementary measure to ensure the quality of a service-based application and thus "can be used to provide additional defense against catastrophic failure" [1]. Several research projects investigating the dependability and quality of services recognize the overlap (commonalities in terms of overall goals and problems) between monitoring and testing (such as WS-DIAMOND[2], PLASTIC[3], or ProTest[4]). However, it is generally argued for a separation between *offline* (development time) techniques such as testing and *online* techniques such as monitoring. Possible synergies which could be exploited if testing was being performed during run-time to augment monitoring are at most subject for future work (e.g., see [39]).

SeCSE is one of the first projects to investigate, in more detail, the combined usage of service monitoring and testing [40, 41].

[2] http://wsdiamond.di.unito.it

[3] http://www.ist-plastic.org

[4] http://www.protest-project.eu

Three ways of combining testing and monitoring activities are introduced. Firstly, SeCSE proposes to monitor the behavior of a service and to statistically analyze the collected monitoring data with the aim of inferring assertions (e.g., invariants).

Secondly, search-based techniques are proposed for test data generation to produce test cases that are likely to violate the SLAs. During test case execution, quality of service (QoS) parameters are observed through monitoring mechanisms, and those QoS parameters are used in turn to guide the search for better test cases. Compared to our envisioned approach the search-based techniques are used to derive "better" test cases and not to gain "higher" confidence in the test results. However, adopting a search-based strategy for producing test cases could promise to be a possible solution for the challenge faced in our approach with respect to test case generation (see Section 4.2).

Thirdly, the use of monitoring data for mimicking service behavior is suggested to reduce the number of required service invocations when executing a test suite. Mimicking the service behavior does not aim at increasing the confidence of the test results, but at reducing the overall effort for testing. Also, as observed in [41] the use of monitoring data for mimicking service behavior is limited to the situations in which the relationship between a service's input and output is deterministic, which, as motivated in Section 1, may not be the case for SBAs.

4. APPROACH

To predict failures with confidence and thereby avoid unnecessary pro-active adaptations, our envisioned solution is based on integrating monitoring and testing techniques for services. This integration enables our approach to exploit available monitoring data, and thus – due to the synergy effects between monitoring and testing – promises to keep the testing effort smaller than if the techniques were applied individually.

Below, we first discuss the assumptions that underlie our approach and then outline the envisioned steps of the approach.

4.1 Assumptions

To initially focus our research activities, we rely on the below assumptions. In Section 5.2, we discuss how those assumptions could be relaxed and addressed in future work.

- *Assumption 1:* We assume that each failure of a constituent service of a SBA instance leads to a failure of that SBA instance. This means that, if a constituent service fails, the SBA instance will deviate from its requirements and thus the need for an adaptation arises. Of course, not each service failure necessarily leads to a requirements violation; e.g., in the example from Section 2.1, an end-to-end performance requirement could still be met if a slower response of the *GMail* service is compensated by a fast response of the *ePay* service. Therefore, there should be an additional step involved in determining whether a failure of a constituent service in fact leads to the need for an adaptation.

- *Assumption 2:* We assume that the observed (monitored or tested) elements provide a notification in case of any change which would invalidate the monitoring or testing data. Examples of such changes are: a new version of the service implementation and a re-deployment of a service. This could be realized, for instance, by means of dedicated service level agreements (SLAs) that bind the service provider to notify the service consumer of changes, or by using special service registries that provide notification functionalities.

- *Assumption 3:* We assume that invoking a constituent service of a SBA for test purposes will have no side effects; e.g., when testing a book delivery service, no books would actually be delivered as a result of the testing activities.

4.2 Steps of the Approach

Our approach prescribes 5 steps to be carried out for determining failure predictions with confidence. Figure 2 depicts those steps.

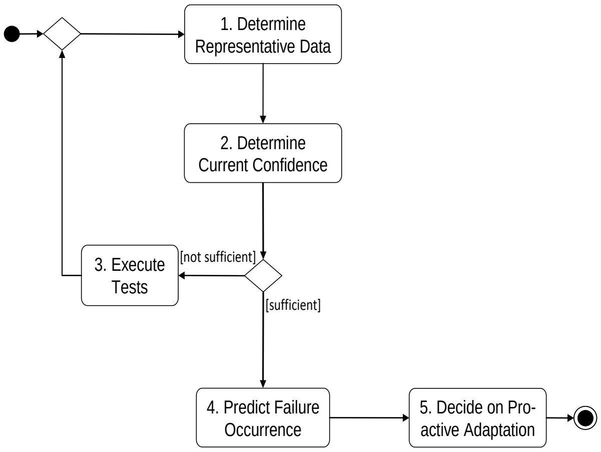

1. Determine Representative Data
2. Determine Current Confidence
3. Execute Tests
[not sufficient]
[sufficient]
4. Predict Failure Occurrence
5. Decide on Pro-active Adaptation

Figure 2: Steps of our solution approach.

It should be noted that during the execution of online tests, the service might have changed or new monitoring data might have been collected (from the SBA instances running in parallel). This means that some of the data will not be representative anymore or that new, representative data should be considered. Thereupon, we follow an incremental process to perform the online tests for attaining the required confidence in failure prediction, i.e., tests will be incrementally executed until the required confidence is reached. In Figure 2 this is reflected by the loop from Step 3 to Step 1.

In the following, we explain the steps of approach and point out some of the challenges that remain to be addressed to realize those steps:

- *Step 1:* The first step determines which of the data points collected so far are representative of the service that is being observed. This is an important step, as in the case of adaptive systems, many data points that have been collected before the change of the application will not be representative of the application after the change. Thus, those data points cannot be considered for assessing the confidence in the failures. In case one is notified of the change of a service (see *Assumption 2* from Section 4.1), one could invalidate the past data points and aim at collecting new ones.

- *Step 2:* The second step determines the confidence of the failure prediction based on the representative data points (from Step 1). As we cannot rely on the assumption that the data is drawn from a given probability distribution, we rely on techniques from non-parametric statistics (e.g., see [42]) for this step. If, the expected confidence is achieved, the approach continues with Step 4. Otherwise, the approach continues with Step 3 in order to collect additional data points.

- *Step 3:* In step 3, test cases are generated (e.g., using [43, 44, 45, 46]) and executed (e.g., using [47, 30]) in order to gather additional, representative data points for failure prediction. Based on the confidence level computed in Step 2, one or more test cases are executed. One key challenge is how to determine adequate test cases that allow achieving the goal of establishing confidence (see Section 3.4).

- *Step 4:* After the previous steps have established a set of representative data that exhibits the required confidence for failure prediction, Step 4 predicts the actual occurrence of the failure. This is done – of course depending on the nature of the failure – using statistical techniques to compute the probability for that failure, or by exploiting specific techniques from data mining [48]. The accuracy of the prediction is related to the technique employed for the the prediction.
- *Step 5:* Step 5 decides on the actual pro-active adaptation of the SBA instances. The decision on such an adaptation is based on the predicted failure probability from Step 4. For example, pro-active adaptation is triggered if the prediction is above a pre-defined threshold. One challenge will be to support more complex decision mechanisms that, for instance, exploit a cost model that takes into account the probability of the failure and its associated costs versus the costs for performing the pro-active adaptation. As an interesting extension of this step, the use of online testing to 'pre-test' the adaptation could be investigated (see [3]). For example, if a service S is to be replaced by service S', S' could first be tested before being bound to the SBA instance.

To initiate the above steps, various strategies can be followed. Below two typical strategies are listed:

- *Strategy A:* Step 1 is triggered as soon as monitoring uncovers a failure. This strategy is efficient in that it only triggers the steps of our approach (and thus online tests) when the potential need for a pro-active adaptation occurs. However, this can lead to delays in adaptation, as some time can be required to run the test cases needed to gain the required confidence for the failure prediction. If those delays are too long, this might even mean that the SBA instances to be adapted already have invoked the service which might had to be replaced.
- *Strategy B:* Step 1 is triggered after each change of a constituent service of the SBA. This continuous update can provide faster reaction times, as more data might be available once an actual failure happens. However, this can lead to increased costs, because some of the produced test results might never be used. As an example, if a service S has been tested and is then modified before a failure of S has occurred, those test results might not be representative for the modified service anymore and thus have been collected unnecessarily.

5. DISCUSSION

5.1 Demonstration of Applicability

This section illustrates how the unnecessary pro-active adaptation as depicted in Section 2.2 can be avoided. Following our approach from above, this means that a confidence in predicting whether the *SecurePay* service will fail during the execution of the yet-to-invoke-SecurePay instances is required.

Let us say that the expected confidence is 95%, and that in order to trigger pro-active adaptation, the failure should occur with a probability of more than 10%.

Figure 3 illustrates some concrete data collected during the execution of the *SecurePay* service (response time) together with the attained confidence levels.

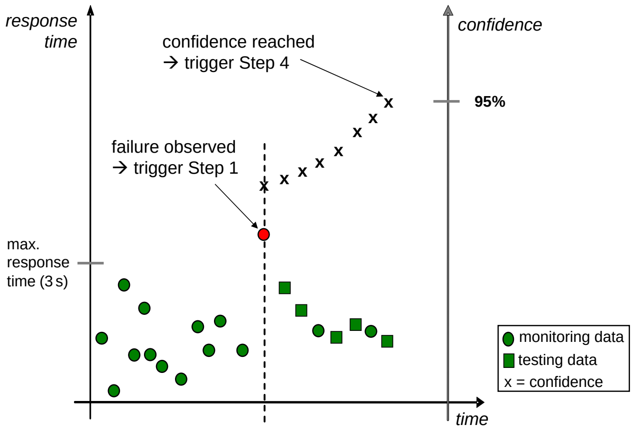


Figure 3: Attaining the required confidence by augmenting monitoring data with test data for the *SecurePay* service.

As can be seen from the figure, the steps of our approach are triggered after the occurrence of the first failure, i.e., we follow Strategy A from Section 4.2. After the failure has occurred, first, representative historical data from past executions of the eGovernment application are determined (Step 1). Based on this data, the confidence is determined (Step 2) and it is checked whether it is sufficient to make the prediction. As can be seen in the example in Figure 3, the confidence computed based on the data available at the time of the failure is not enough, and thus further data points are collected, until the confidence level of 95% is reached.

As depicted in Figure 3, dedicated tests are incrementally executed against the *SecurePay* service in such a way that test results complement the historical data (Step 3). Please note that in parallel to those tests, additional monitoring data is collected, which is considered to compute the confidence.

Once the confidence level of 95% has been reached, the probability for the failure is computed (Step 4). Using the relative frequency of the ocurrence of a failure as a naive measure of the failure probability, a 'probability' of 5% (1 out of 20) in the example can be determined. This is below the threshold of 10%. Accordingly, no pro-active adaptation will be triggered (Step 5).

5.2 Relaxation of Assumptions

Below, we discuss how the assumptions from Section 4.1 can be relaxed:

- *Assumption 1: 'Each failure of a constituent service of a SBA instance leads to a failure of that SBA instance.'* As mentioned above, a failure of a constituent services does not necessarily imply a deviation of the SBA instance from its requirements. Thus, upon discovery of a failure, an additional step is required to determine such a requirements violation. In [49], we have introduced a technique based on run-time verification, which could be exploited to that end. If Strategy A (from Section 4.2) is employed, addressing this assumption provides a very good way for further reducing the effort needed for online testing. Only if the observed failure can lead to a requirements deviation, an adaptation of the SBA instance needs to be performed and thus online tests might need to be initiated to gather additional data points for the expected confidence.

- *Assumption 2: 'The observed (monitored or tested) elements provide a notification in case of any change which would invalidate the monitoring or testing data.'* In the case that this assumption does not hold (e.g., because there are no SLAs that would bind the service provider in notifying the consumers about changes), techniques from data mining (e.g., such as [50]) could be applied to determine a change of the observed entity.

- *Assumption 3: 'Invoking the constituent services of a SBA for test purposes will have no side effects.'* This assumption can be relaxed by ensuring that a constituent service will either only be queried (i.e., provide a response without any side-effects), or will provide a dedicated test mode (e.g., as suggested by [51]). This is not trivial for example, when considering performance issues; e.g., the high load of test cases can have impact on the load of the service hardware.

6. CONCLUSIONS AND PERSPECTIVES

The position we took in this paper is that augmenting monitoring data with results from dedicated online tests promises to produce failure predictions with confidence. The failure predictions can be taken into account during the actual adaptation decisions. This allows deciding with high certainty whether to adapt service-based applications, thereby avoiding unnecessary pro-active adaptations of those applications. In addition, due to the synergy effects between monitoring and testing, our approach promises to keep the testing effort smaller than if the techniques were applied in isolation.

The contribution of this paper has been to motivate our approach and support its novelty by means of a thorough discussion of the related work. We have demonstrated the application of our approach by using an example scenario from the eGovernment domain.

In addition, we have identified challenges that need to be addressed to implement our approach. One challenge will be to support more complex pro-active adaptation decision mechanisms that, for instance, exploit a cost model that takes into account the probability of the failure and its associated costs versus the costs for performing the pro-active adaptation. Another key challenge is how to determine and execute adequate test cases that allow achieving the goal of establishing confidence in failure predictions while taking into account the imapct of the tests execution on the load of the service hardware. Those research challenges are currently addressed in the European Network of Excellence on Software Services and Systems (S-Cube[5]).

Acknowledgments

The authors are grateful to the reviewers of the paper for their fruitful comments.

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

7. REFERENCES

[1] Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Software Eng. **30** (2004) 859–872

[2] Benbernou, S.: State of the art report, gap analysis of knowledge on principles, techniques and methodologies for monitoring and adaptation of sbas. Deliverable PO-JRA-1.2.1, S-Cube Consortium (2008) http://www.s-cube-network.eu/.

[3] Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A framework for proactive self-adaptation of service-based applications based on online testing. In: ServiceWave 2008. Number 5377 in LNCS, Springer (2008)

[4] Nitto, E.D., Mazza, V., Mocci, A.: Collection of industrial best practices, scenarios and business cases. Deliverable CD-IA-2.2.2, S-Cube Consortium (2009) http://www.s-cube-network.eu/.

[5] Sahoo, R.K., Oliner, A.J., Rish, I., Gupta, M., Moreira, J.E., Ma, S., Vilalta, R., Sivasubramaniam, A.: Critical event prediction for proactive management in large-scale computer clusters. In: KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, ACM (2003) 426–435

[6] Liang, Y., Zhang, Y., Sivasubramaniam, A., Jette, M., Sahoo, R.: Bluegene/l failure analysis and prediction models. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society (2006) 425–434

[7] Song, H., Leangsuksun, C.b., Nassar, R.: Availability modeling and analysis on high performance cluster computing systems. In: ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security, Washington, DC, USA, IEEE Computer Society (2006) 305–313

[8] Fu, S., Xu, C.Z.: Exploring event correlation for failure prediction in coalitions of clusters. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (2007) 1–12

[9] Fu, S., Xu, C.Z.: Quantifying temporal and spatial correlation of failure events for proactive management. In: SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, Washington, DC, USA, IEEE Computer Society (2007) 175–184

[10] Fox, A., Kiciman, E., Patterson, D.: Combining statistical monitoring and predictable recovery for self-management. In: WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, New York, NY, USA, ACM (2004) 49–53

[11] Myers, G.: The Art of Software Testing. Wiley (2004)

[12] McGregor, J., Sykes, D.: A Practical Guide to Testing Object-oriented Software. Addison-Wesley Professional (2001)

[13] Osterweil, L.J.: Strategic directions in software quality. ACM Comput. Surv. **28** (1996) 738–750

[14] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall (1991)

[15] Baresi, L., Nitto, E.D., eds.: Test and Analysis of Web Services. Springer (2007)

[16] Pernici, B., Metzger, A.: Survey of quality related aspects relevant for service-based applications. Deliverable PO-JRA-1.3.1, S-Cube Consortium (2008) http://www.s-cube-network.eu/.

[17] Poore, J., Trammell, C.: Engineering practices for statistical testing. Crosstalk: The Journal of Defense Software Engineering **11** (1998) 24–28

[5] http://www.s-cube-network.eu/

[18] Trammell, C.: Quantifying the reliability of software: statistical testing based on a usage model. In: ISESS '95: Proceedings of the 2nd IEEE Software Engineering Standards Symposium, Washington, DC, USA, IEEE Computer Society (1995) 208

[19] Whittaker, J.A., Thomason, M.G.: A markov chain model for statistical software testing. IEEE Trans. Softw. Eng. **20** (1994) 812–824

[20] Bauer, T., Bohr, F., Landmann, D., Beletski, T., Eschbach, R., Poore, J.: From requirements to statistical testing of embedded systems. In: SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems, Washington, DC, USA, IEEE Computer Society (2007) 3

[21] Kallepalli, C., Tian, J.: Measuring and modeling usage and reliability for statistical web testing. IEEE Trans. Softw. Eng. **27** (2001) 1023–1036

[22] Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering (2008)

[23] Wang, Q., Quan, L., Ying, F.: Online testing of Web-based applications. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC). (2004) 166–169

[24] Deussen, P., Din, G., Schieferdecker, I.: A TTCN-3 based online test and validation platform for Internet services. In: Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS). (2003) 177–184

[25] Chan, W., Cheung, S., Leung, K.: A metamorphic testing approach for online testing of service-oriented software applications. International Journal of Web Services Research **4** (2007) 61–81

[26] Bai, X., Chen, Y., Shao, Z.: Adaptive web services testing. In: 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 233–236

[27] Bai, X., Xu, D., Dai, G., Tsai, W., Chen, Y.: Dynamic reconfigurable testing of service-oriented architecture. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 368–375

[28] Ruth, M., Oh, S., Loup, A., Horton, B., Gallet, O., Mata, M., Tu, S.: Towards automatic regression test selection for web services. In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC). (2007) 729–734

[29] Ruth, M., Tu, S.: A safe regression test selection technique for Web services. In: Second International Conference on Internet and Web Applications and Services (ICIW). (2007)

[30] Di Penta, M., Bruno, M., Esposito, G., et al.: Web Services Regression Testing. In Baresi, L., Di Nitto, E., eds.: Test and Analysis of Web Services. Springer (2007) 205 – 234

[31] F. Nielson, H. R. Nielson, C.H.: Principles of Program Analysis. Springer (2005) Second Ed.

[32] O'Connor, P., Newton, D., Bromley, R.: Practical reliability engineering. John Wiley & Sons Inc (2002)

[33] Goel, A.L.: Software reliability models: Assumptions, limitations, and applicability. IEEE Trans. Softw. Eng. **11** (1985) 1411–1423

[34] Zhang, F., Zhou, X., Chen, J., Dong, Y.: A novel model for component-based software reliability analysis. In: HASE '08: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium, Washington, DC, USA, IEEE Computer Society (2008) 303–309

[35] Roshandel, R., Medvidovic, N., Golubchik, L.: A bayesian model for predicting reliability of software systems at the architectural level. In: QoSA. (2007) 108–126

[36] Roshandel, R., Banerjee, S., Cheung, L., Medvidovic, N., Golubchik, L.: Estimating software component reliability by leveraging architectural models. In: ICSE. (2006) 853–856

[37] Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L.: Early prediction of software component reliability. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 111–120

[38] Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2009) 111–121

[39] —: Ws-diamond: Web services - diagnosability, monitoring and diagnosis: Publishable final activity report. Technical report, WS-DIAMOND EU FP6 protect (2008)

[40] Canfora, G., Di Penta, M.: Testing services and service-centric systems: challenges and opportunities. IT Professional **8** (2006) 10–17

[41] —: Testing method definition. deliverable A1.D3.4. Technical report, SeCSE EU FP6 Project (2008)

[42] Corder, G., Foreman, D.: Nonparametric statistics for non-statisticians: A step-by-step approach. Wiley-Blackwell (2009)

[43] Bai, X., Dong, W., Tsai, W.T., Chen, Y.: WSDL-Based Automatic Test Case Generation for Web Services Testing. In: Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE), IEEE Computer Society (2005) 215 – 220

[44] Tarhini, A., Fouchal, H., Mansour, N.: A simple approach for testing Web service based applications. In: 5th International Workshop on Innovative Internet Community Systems. Lecture Notes in Computer Science Vol.3908 (2006) 134–146

[45] Lübke, D.: Unit Testing BPEL Compositions. In Baresi, L., Di Nitto, E., eds.: Test and Analysis of Web Services. Springer (2007) 149 – 171

[46] Dong, W.L., Yu, H., Zhang, Y.B.: Testing BPEL-based Web Service Composition Using High-level Petri Nets. In: EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, IEEE Computer Society (2006) 441–444

[47] Martin, E., Basu, S., Xie, T.: Automated Testing and Response Analysis of Web Services. In: IEEE International Conference on Web Services (ICWS). (2007) 647 – 654

[48] Xue, Z., Dong, X., Ma, S., Dong, W.: A survey on failure prediction of large-scale server clusters. Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS International Conference on **2** (2007) 733–738

[49] Gehlert, A., Bucchiarone, A., Kazhamiakin, R., Metzger, A., Pistore, M., Pohl, K.: Exploiting assumption-based verification for the adaptation of service-based applications. In: Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), NNN, 2010. (2010)

[50] Bickel, S., Brückner, M., Scheffer, T.: Discriminative learning under covariate shift. Journal of Machine Learning Research **10** (2009) 2137–2155

[51] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R.: The MORABIT approach to runtime component testing. In: Proceedings of the 30th Annual Int'l Computer Software and Applications Conference (COMPSAC). (2006) 171–176

Runtime Prediction of Service Level Agreement Violations for Composite Services

Philipp Leitner[1], Branimir Wetzstein[2], Florian Rosenberg[3], Anton Michlmayr[1], Schahram Dustdar[1], Frank Leymann[2]

[1] Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040, Vienna, Austria
lastname@infosys.tuwien.ac.at

[2] Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

[3] CSIRO ICT Centre
GPO Box 664
Canberra ACT 2601, Australia
florian.rosenberg@csiro.au

Abstract. SLAs are contractually binding agreements between service providers and consumers, mandating concrete numerical target values which the service needs to achieve. For service providers, it is essential to prevent SLA violations as much as possible to enhance customer satisfaction and avoid penalty payments. Therefore, it is desirable for providers to predict possible violations before they happen, while it is still possible to set counteractive measures. We propose an approach for predicting SLA violations at runtime, which uses measured and estimated facts (instance data of the composition or QoS of used services) as input for a prediction model. The prediction model is based on machine learning regression techniques, and trained using historical process instances. We present the architecture of our approach and a prototype implementation, and validate our ideas based on an illustrative example.

1 Introduction

In service-oriented computing [1], finer-grained basic functionality provided using Web services can be composed to more coarse-grained services. This model is often used by Software-as-as-Service providers to implement value-added applications, which are built upon existing internal and external Web services. Very important for providers and consumers of such services are Service Level Agreements (SLAs), which are legally binding agreements governing the quality that the composite service is expected to provide (Quality of Service, QoS) [2]). SLAs contain Service Level Objectives (SLOs), which are concrete numerical target values (e.g., "maximum response time is 45 seconds"). For the provider it is essential to not violate these SLOs, since typically violations are coupled with penalty payments. Additionally, violations can negatively impact service consumer satisfaction. Therefore, it is vitally important for the service provider to be aware of SLA violations, in order to react to them accordingly.

Typically, SLA monitoring is done *ex post*, i.e., violated SLOs can only be identified after the violation happened. While this approach is useful in that it alerts the provider to potential quality problems, it clearly cannot directly help preventing them. In that regard an *ex ante* approach is preferable, which allows to predict possible SLA violations before they have actually occurred. The main contribution of this paper is the introduction of a general approach to prediction of SLA violations for composite services, taking into account both QoS and process instance data, and using estimates to approximate not yet available data. Additionally, we present a prototype implementation of the system and an evaluation based on an order processing example. The ideas presented here are most applicable for long-running processes, where human intervention into problematic instances is possible. Our system introduces the notions of checkpoints (points in the execution of the composition where prediction can be done), facts (data which is already known in a checkpoint, such as the response times of already used services) and estimates (data which is not yet available, but can be estimated). Facts and estimates can refer to both typical QoS data (e.g., response times, availability, system load) and process instance data (e.g., customer identifiers, ordered products). Our implementation uses regression classifiers, a technique from the area of machine learning [3], to predict concrete SLO values.

The rest of the paper is structured as follows. In Section 2 we briefly introduce an illustrative example which we will use in the remainder of the paper. In Section 3 we detail the general concepts of our prediction approach. In Section 4 we described the implementation of a prototype tool, which we use for evaluation in Section 5. Finally, we provide an overview of relevant related work in Section 6 and conclude the paper in Section 7.

2 Illustrative Example

To illustrate the ideas presented in this paper we will use a simple purchase order scenario (see Figure 2 below). In this example there are a number of roles to consider: a reseller, who is the owner of the composite service, a customer, who is using it, a banking service, a shipping service, and two external suppliers. The business logic of the reseller service is defined as follows. Whenever the reseller service receives an order from the customer, it first checks if all ordered items are available in the internal stock. If this is not the case, it checks if the missing item(s) can be ordered from Supplier 1, and, if this is not the case, from Supplier 2. If both cannot deliver the order has to be cancelled, otherwise the missing items are ordered from the respective supplier. When all ordered items are available she will (in parallel) proceed to charge the customer using the banking service and initialize shipment of the ordered goods (using the Shipping

Service). We have borrowed this example from [4], please refer to this work for more information.

In this case study, the reseller has an SLA with its customers, with an SLO specifying that the end-to-end response time of the composition cannot be more than a certain threshold of time units. For every time the SLO is violated the customer is contractually entitled a discount for the order. Note that even though our explanations in this paper will be based on just one single SLO, our approach can be generalized to multiple SLOs. Additionally, even though we present our approach based on a numerical SLO, our ideas can be also applied to estimation of nominal objectives. However, SLOs need to adhere to the following requirements: (1) they need to be non-deterministic (following the definition in [5]), and (2) they cannot be defined as aggregations over multiple executions. Requirement (1) is not so much functionally important, but our prediction approach is not very useful otherwise (e.g., for SLOs concerning security requirements). Requirement (2) is a limitation of our current approach, which we plan on working on as part of our future work.

3 Predicting SLA Violations

In this section we present the core ideas of our approach towards prediction of SLA violations. Generally, the approach is based on the idea of predicting concrete SLO values based on whatever information is already available at a concrete point in the execution of a composite service. We distinguish three different types of information. (1) **Facts** represent data which is already known at prediction time. Typical examples of facts are the QoS of already used services, such as the response time of a service which has already been invoked in this execution, or instance data which has either been passed as input or which has been generated earlier in the process execution. (2) **Unknowns** are the opposites of facts, in that they represent data which is entirely unknown at prediction time. Oftentimes, instance data which has not yet been produced falls into this category. If important factors are unknown at prediction time the prediction quality will be very bad, e.g., in our illustrative example a prediction cannot be accurate before it is known whether the order can be delivered from the reseller's internal stock. (3) **Estimates** are a kind of middle ground between facts and unknowns, in that they represent data which is not yet available, but can be estimated. This is often the case for QoS data, since techniques such as QoS monitoring [5] can be used to get an idea of e.g., the response time of a service before it is actually invoked. Estimating instance data is more difficult, and generally domain-specific.

The overall architecture of our system is depicted in Figure 1. The most important concept used is that the user defines **checkpoints** in the service composition, which indicate points in the execution where a prediction should be carried out. The exact point in the execution model which triggers the checkpoint is called the **hook**. Every checkpoint is associated with one **checkpoint predictor**. Essentially, the predictor uses a function taking as input all facts

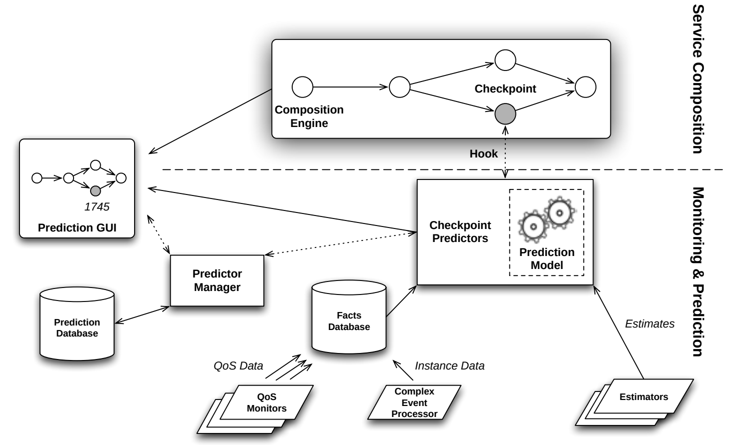


Fig. 1: Overall System Architecture

which are already available in the checkpoint, and, if applicable, a number of estimates of not yet known facts, and produces a numerical estimation of the SLO value(s). This function is generated using machine learning techniques We refer to this function as the **prediction model** of a checkpoint predictor. Facts are retrieved from a **facts database**, which is filled using a number of **QoS monitors** (which provide QoS data) and a **Complex Event Processing** (CEP) engine (which extracts and correlates the instance data, as emitted by the process engine). A detailed discussion of our event-based approach to monitoring is out of scope of this paper, but can be reviewed in related work [4, 6]. **Estimators** are a generic framework for components which deliver estimates. Finally, the prediction result is transferred to a graphical user interface (**prediction GUI**), which visualizes the predicted value(s) for the checkpoint. A **predictor manager** component is responsible for the lifecycle management of predictors, i.e., for initializing, destroying and retraining them. Additionally, predictions are stored in a **prediction database** to be available for future analysis.

3.1 Checkpoint Definition

At design-time, the main issue is the definition of checkpoints in the composition model. For every checkpoint, the following input needs to be provided: (1) The hook, which defines the concrete point in the execution that triggers the prediction, (2) a list of available facts, (3) a list of estimates, and the estimator component as well as the parameters used to retrieve or calculate them, (4) the retraining strategy, which governs at which times a rebuilding of the prediction model should happen, and (5) as a last optional step, a parameterization of the machine learning technique used to build the prediction model. After all these inputs are defined the checkpoint is deployed using the predictor manager, and an initial model is built. For this a set of historical executions of the composite service need to be available, for which all facts (including those associated with estimates) have been monitored. If no or too little historical data is available the checkpoint is suspended by the predictor manager until enough training data has been collected. The amount of data necessary is case-specific, since it vastly depends on the complexity of the composition. We generally use the Training Data Correlation as a metric for evaluating the quality of a freshly trained model (see below for a definition), however, a detailed discussion of this is out of scope of this paper. After the initial model is built the continuous optimization of the predictor is governed by the predictor manager, according to the retraining strategy. Finally, the checkpoint can be terminated by the user via the prediction GUI. We will now discuss these concepts in more depth.

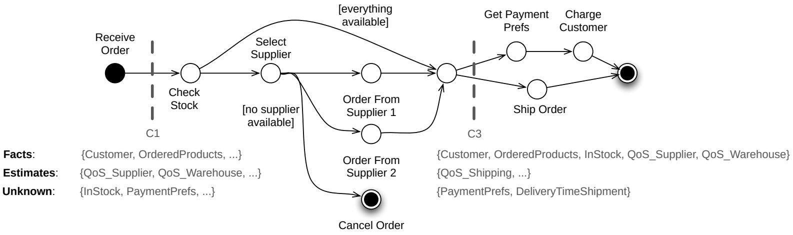

Receive Order
[everything available]
Select Supplier
Get Payment Prefs
Charge Customer
Check Stock
[no supplier available]
Order From Supplier 1
Order From Supplier 2
Ship Order
Cancel Order
C1
C3
Facts: {Customer, OrderedProducts, ...}
Estimates: {QoS_Supplier, QoS_Warehouse, ...}
Unknown: {InStock, PaymentPrefs, ...}
{Customer, OrderedProducts, InStock, QoS_Supplier, QoS_Warehouse}
{QoS_Shipping, ...}
{PaymentPrefs, DeliveryTimeShipment}

Fig. 2: Illustrative Example With Possible Checkpoints

Hooks Hooks can be inserted either before or after any WS-BPEL activity (for instance, an *Invoke* activity). Generally, there is a tradeoff to take into account here, since early predictions are usually more helpful (in that they rather allow for corrections if violations are predicted), but also less accurate since less facts are available and more estimates are necessary. Figure 2 depicts the (simplified) example from Section 2, and shows two possible checkpoints. In C_1 the only facts available are the ones given as input to the composition (such as a customer identifier, or the ordered products). Some other facts (mainly QoS metrics) can already be estimated, however, other important information, such as whether the order can be served directly from stock, is simply unavailable in C_1, not even as an estimate. Therefore, the prediction cannot be very accurate. In checkpoint C_3, on the other hand, most of the processes important raw data is already available as facts, allowing for good predictions. However, compared to C_1, the possibilities to react to problems are limited, since only the payment and shipping steps are left to adapt (e.g., a user may still decide to use express shipping instead of the regular one if a SLA violation is predicted in C3). Finding good checkpoints at which the prediction is reasonably accurate and still timely enough to react to problems demands for some domain knowledge about influential factors of composition performance. Dependency analysis as discussed in [6] can help providing this crucial information. Dependency analysis is the process of using historical business process instance data to find out about the main factors which dictate the performance of a process. When defining checkpoints, a user can assume that the majority of important factors of influence need to be available as either facts or at least as good estimates in order to achieve accurate predictions.

Facts and Estimates: Facts represent all important information which can already be measured in this checkpoint. This includes both QoS and instance data. Note that the relationship between facts and the final SLO values does not need to be known (e.g., a user can include instance data such as user identifiers or ordered items, even if she is not sure if this has any relevance for the SLO). However, dependency analysis can again be used to identify the most important facts for a checkpoint. Additionally, the user can also define estimates. In the example above, in C_1 the response time of the warehouse service is not yet known, however, it can e.g., be estimated using a QoS monitor. Since estimating instance data is inherently domain-specific, our system is extensible in that more specific estimators (which are implemented as simple Java classes) can be integrated seamlessly. Estimates are linked to facts, in the sense that they have to represent an estimation of a fact which will be monitorable at a later point.

Retraining Strategy: Generally, the prediction model needs to be rebuilt whenever enough new information is available to significantly improve the model. The retraining strategy is used to define when the system should check whether rebuilding the prediction model is necessary. Table 1 summarizes all retraining strategies available, and gives examples. The custom strategy is defined using Java code, all other strategies are implemented in our prototype and can be used and configured without any additional code.

Strategy	**Retrains ...**	**Example**
`periodic`	... in fixed intervals	*every 24 hours*
`instance-based`	... whenever a fixed number of new instances have been received since the last training	*every 250 instances*
`on demand`	... on user demand	–
`on error`	... if the mean prediction error exceeds a given threshold	*if* $\bar{e} > T$
`custom`	... if a user-defined condition applies	*whenever more than 10 orders from customer 12345 have been received*

Table 1: Predictor Retraining Strategies

Prediction Model Parameterization: A user can also define the machine learning technique that should be used to build the prediction model. This is done by specifying an algorithm and the respective parameterization for the WEKA toolkit[4], an open source machine learning toolkit which we internally use in our prototype implementation. In this way the prediction quality can be tuned by a machine learning savvy user, however, we also provide a default configuration which can be used out of the box.

3.2 Run-Time Prediction

At runtime, the prediction process is triggered by lifecycle events from the WS-BPEL engine. These are events emitted by some engines (such as Apache ODE[5]), which contain lifecycle information about the service composition (e.g., `ActivityExecStartEvent`, `VariableModificationEvent`, `ProcessCompletionEvent`). Our approach is based on these events, therefore, a WS-BPEL engine which is capable of emitting these events is a preliminary of our approach. When checkpoints are deployed we use the hook information to register respective event listeners. For instance, for a checkpoint with the hook "After invoke CheckStock" we generate a listener for `ActivityExecEndEvent`s which consider the invoke activity "CheckStock". We show the sequence of actions which is triggered as soon as such an event is received in Figure 3.

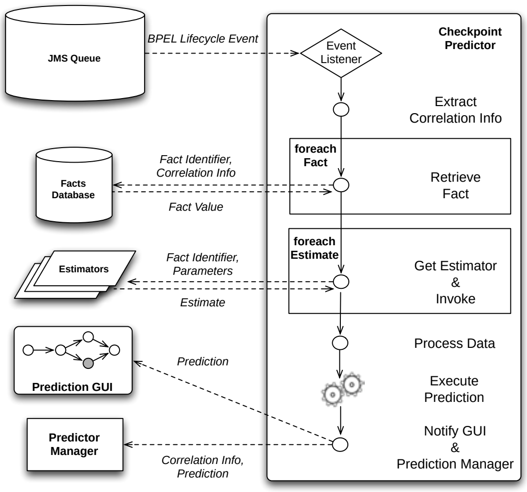
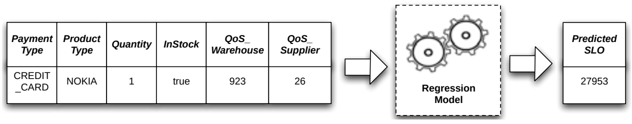


Fig. 3: Runtime View On Checkpoint Predictors

After being triggered by a lifecycle event the checkpoint predictor first extracts some necessary correlation information from the event received. This in-

[4] `http://www.cs.waikato.ac.nz/ml/weka/`

[5] `http://ode.apache.org`



Payment Type	*Product Type*	*Quantity*	*InStock*	*QoS_ Warehouse*	*QoS_ Supplier*
CREDIT _CARD	NOKIA	1	true	923	26

Predicted SLO
27953

Fig. 4: Black-Box Prediction Model

cludes the process instance ID as assigned by the composition engine, the instance start time (i.e., the time when the instance was created) and the timestamp of the event. This information is necessary to be able to retrieve the correct facts from the facts database, which is done for every fact in the next step (e.g., in order to find the correct fact "CustomerNumber" for the current execution the process instance ID needs to be known). When all facts have been gathered, the predictor also collects the still missing estimates. For this, for every estimate the predictor instantiates the respective estimator component (if no instance of this estimator was available before), and invokes it (passing all necessary parameters as specified in the checkpoint definition). The gathered facts and estimates are then converted into the format expected by the prediction model (in the case of our prototype, this is the WEKA Attribute-Relation File Format ARFF[6]), and, if necessary, some data cleaning is done. Afterwards, the actual prediction is carried out by passing the gathered input to the prediction model producing a numerical estimation of the SLO value. This prediction is then passed to the prediction GUI (for visualization) and the prediction manager.

Note that the "intelligence" that actually implements the prediction of the SLO values is encapsulated in the prediction model. Since we (usually) want to predict numerical SLO values the prediction model needs to be a regression model [3]. We consider the regression model to be a black-box function which takes a list of numeric and nominal values as input, and produces a numeric output (Figure 4). Generally, our approach is agnostic of how this is actually implemented. In our prototype we use multilayer perceptrons (a powerful variant of neural networks) to implement the regression model. Multilayer perceptrons are trained iteratively using a back-propagation technique (maximization of the correlation between the actual outcome of training instances and the outcome that the network would predict on those instances), and can (approximately) represent any relationship between input data and outcome (unlike simpler neural network techniques such as the perceptron, which cannot distinguish data which is not separable by a hyperplane [7]). If a non-numerical SLO should be predicted, a different technique suitable for classification (as opposed to regression) needs to be used to implement the prediction model, e.g., decision trees such as C4.5 [8].

[6] `http://www.cs.waikato.ac.nz/~ml/weka/arff.html`

3.3 Evaluation of Predictors

Another important task of the prediction manager is quality management of predictors, i.e., continually supervising how predictions compare to the actual SLO values once the instance is finished. Generally, we use three different quality metrics to measure the quality of predictions in checkpoints, which are summarized in Table 2. The first metric, *Training Data Correlation*, is a standard machine learning approach to evaluating regression models. We use it mainly to evaluate freshly generated models, when no actual predictions have yet been carried out. This metric is defined as the statistical correlation between all training instance outcomes and the predictions that the model would deliver for these training instances. The definition given in the table is the standard statistical definition of the correlation coefficient between a set of predicted values P and a set of measured values M. However, note that this metric is inherently overconfident in our case, since during training all estimates are replaced for the facts that they estimate (i.e., the training is done as if all estimates were perfect). Therefore, we generally measure the prediction error later on, when actual estimates are being used. However, a low training data correlation is an indication that important facts are still unknown in the checkpoint, i.e., that the checkpoint may be too early.

Name	**Definition**
`Training Data Correlation`	$corr = \frac{cov(P,M)}{\sigma_p \sigma_m}$
`Mean Prediction Error`	$\bar{e} = \frac{\sum_{i=0}^{n} \lvert m_i - p_i \rvert}{n}$
`Prediction Error Standard Deviation`	$\sigma = \sqrt{\frac{\sum_{i=0}^{n}(e_i - \bar{e})^2)}{n}}$

Table 2: Predictor Quality Metrics

This can be done using the *Mean Prediction Error* $\bar{e}$, which is the average (Manhatten) difference between predicted and monitored values. In the definition in Table 2, n is the total number of predictions, p_i is a predicted value, and m_i is the measured value to prediction p_i. Finally, we use the *Prediction Error Standard Deviation* (denoted here simply as σ) to describe the variability of the prediction error (i.e., high σ essentially means that the actual error for an instance can be much lower or higher than $\bar{e}$). In the definition, e_i is the actual prediction error for a process instance ($m_i - p_i$). These metrics are mainly used to give the user an estimation of how trustworthy a given prediction is. Additionally, the `on error` retraining strategy triggers on $\bar{e}$ exceeding a certain threshold.

4 Tool Implementation

In order to verify our approach we built a prototype prediction tool in the Java programming language. Our core implementation is based on our earlier work on event-based monitoring and analysis (as presented in [6] and [4]). Data persistence is provided using a simple MySQL[7] database and Hibernate[8]. We have integrated two different approaches to QoS monitoring: firstly, QoS data as provided by the event-based QoS monitoring approach discussed in [6], and secondly, the QoS data provided by server- and client-side VRESCo [9] QoS monitors [5]. In order to enable event-based monitoring we have used Apache ActiveMQ[9] as JMS middleware. Finally, as has already been discussed, we use the open-source machine learning toolkit WEKA to build prediction models. WEKA is integrated in our system using the WEKA Java API. In addition to the actual prediction tool we have also prototypically implemented the illustrative example as presented in Section 2, as a testbed to verify our ideas (this will be discussed in more detail in Section 5). We have used the WS-BPEL engine Apache ODE, mainly because of ODE's strong support for BPEL lifecycle events. We have also set up the necessary base services which are used in the example (e.g., supplier services, banking service, stock service) using Apache CXF[10].

```
<cpdl:checkpoints
  xmlns:cpdl="http://www.infosys.tuwien.ac.at/2009/cpdl">

  <checkpoint
    name="beforeGetPaymentPrefs"
    activityName="getPaymentPrefs"  breakBefore="true"
    predictor="weka.classifiers.functions.MultilayerPerceptron">

    <update type="periodically" value="5"/>
    <class ppmRef="ORDER_FULFILLMENT_LEAD_TIME"/>
    <fact ppmRef="RESPONSE_TIME_WAREHOUSE"/>
    <fact ppmRef="ORDER_INSTOCK"/>
    <!-- more facts -->
    <estimate name="getPaymentPrefsResponseTime" type="integer">
      <estimatorClass
        class="at.ac.tuwien.infosys.branimon.VrescoQoSEstimator"/>
      <argument value="ResponseTime"/>
      <argument value="CustomerService"/>
      <estimatedField ppmRef="RESPONSE_TIME_GETPAYMENTPREFS"/>
    </estimate>

    <!-- more estimates -->
  </checkpoint>

<checkpoints>
```

Fig. 5: Checkpoint Definition in XML Notion

[7] http://www.mysql.com/

[8] https://www.hibernate.org/

[9] http://activemq.apache.org/

[10] http://cxf.apache.org/

As discussed in Section 3, the main input for our approach is a list of checkpoint definitions. In our current prototype, these definitions are given in a proprietary XML-based language, which we refer to as CPDL (Checkpoint Definition Language). An exemplary excerpt can be seen in Figure 5. In the figure, a checkpoint, which is hooked before the execution of the invoke activity "getPaymentPrefs", is defined. A multilayer perceptron is used as prediction model. The checkpoint will be retrained periodically every 5 hours, and will predict the SLO `ORDER_FULFILLMENT_LEAD_TIME`. Then a number of available facts and estimates are specified. For estimates, an estimator class is given as a full qualified Java class name, which implements the actual prediction. Additionally, a number of arguments can be given to the estimator class. Finally, for every estimate a link to the estimated fact needs to be specified. Note that we do not define facts directly in CPDL. Instead, we reuse the model presented in [6], where we discussed a language for definition of facts using calculation formulae and XLink[11] pointers to WS-BPEL processes (so-called PPMs, process performance metrics). In CPDL, `ppmRefs` are identifiers which point to PPMs in such a model. The complete XML Schema definition of CPDL is available online[12].

5 Experimentation

In order to provide a first validation of the ideas presented we have implemented the illustrative example as discussed in Section 2, and run some experiments using our prototype tool. All experiments have been conducted on a single test machine with 3.0 GHz and 32 GByte RAM, running under Windows Server 2007 SP1. We have repeated every experiment 25 times and averaged the results, to reduce the influence of various random factors such as current CPU load or workload of the process engine.

Instances	**Training [ms]**
100	3545
250	8916
500	17283
1000	31806

(a) Training Overhead

Instances	**Prediction [ms]**
100	615
250	630
500	631
1000	647

(b) Prediction Overhead

Table 3: Overhead for Training and Prediction

In Table 3 we have sketched the measured times for two essential operations of our system. Table 3(a) depicts the amount of time in milliseconds necessary to build or refresh a prediction model in a checkpoint. The most important factor here is clearly the time necessary to train the machine learning model,

[11] `http://www.w3.org/TR/xlink/`

[12] `http://www.infosys.tuwien.ac.at/staff/leitner/cpdl/cpdl_model.xsd`

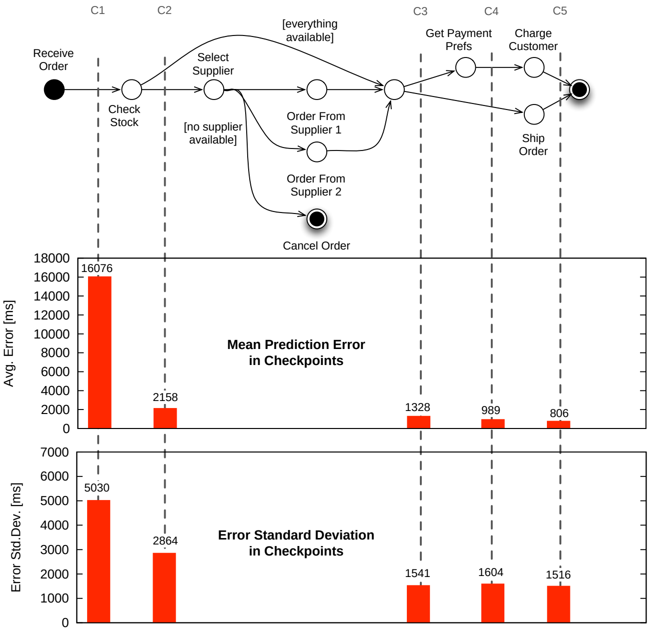


Fig. 6: Prediction Error in Checkpoints

e.g., to train the neural network in our illustrative example. This factor mainly depends on the number of training instances available. In Table 3(a) it can be seen that the time necessary for building the model depends linearly on the number of historical instances available. However, even for e.g., 1000 instances the absolute rebuilding time is below 32 seconds, which seems acceptable for practice, considering that model rebuilding can be done sporadically and offline. Additionally, when rebuilding the model, there is no time where no prediction model is available at all. Instead, the new model is trained offline, and exchanged for the last model as soon as training is finished. A more detailed discussion of these factors is out of scope of this paper for reasons of brevity. In Table 3(b) we have sketched the time necessary for actual prediction, i.e., the online part of the system. As can be seen this overhead is constant and rather small (well below 1 second), which seems very acceptable for prediction at run-time.

Even more important than the necessary time is the accuracy of predictions. To measure prediction accuracy, we have realized five checkpoints in the illustrative example (see top of Figure 6): C1 is located directly after the order is received, C2 after the internal warehouse is checked, C3 after eventual orders from external suppliers have been carried out, C4 during the payment and shipment process, and finally C5 when the execution is already finished. In each of those checkpoints we have trained a prediction model using 1000 historical process instances, and have specified all available data as facts. For not yet available

QoS metrics we have used the average of all previous invocations as estimate. Missing instance data has been treated as unknown. We have used each of those checkpoints to predict the outcome of 100 random executions, and calculated the Mean Prediction Error $\bar{e}$ and the Error Standard Deviation σ (both as defined in Section 3). As expected, $\bar{e}$ is decreasing with the amount of factual data available. In C1, the prediction is mostly useless, since no real data except the user input is available. However, in C2 the prediction is already rather good. This is mostly due to the fact that in C2 the information whether the order can be delivered directly from stock is already available. In C3, C4 and C5 the prediction is continually improving, since more actual QoS facts are available, and less estimates are necessary. Speaking in absolute values, $\bar{e}$ in e.g., C3 is 1328 ms. Since the average SLO value in our illustrative example was about 16000 ms, the error represents only about 8% of the actual SLO value, which seems satisfactory. Similar to $\bar{e}$, σ is also decreasing, however, we can see that the variance is still rather high even in C3, C4 and C5. This is mostly due to our experimentation setup, which included the (realistic) simulation of occasional outliers, which are generally unpredictable.

6 Related Work

The work presented in this paper is complementary to the more established concept of SLA management [10]. SLA management incorporates the definition and monitoring of SLAs, as well as the matching of consumer and provider templates. [10] introduces SLA management based on the WSLA language. However, other possibilities exist, e.g., in [11] the Web Service Offerings Language (WSOL) has been introduced. WSOL considers so-called Web service offerings, which are related to SLAs. Runtime management for WSOL, including monitoring of offerings, has been described in [12], via the WSOI management infrastructure. In our work we add another facet to this, namely the prediction of SLA violations before they have actually occurred. Inherently, this prediction demands for some insight into the internal factors impacting composite service performance. In [13], the MoDe4SLA approach has been introduced to model dependencies of composite services on the used base services, and to analyze the impact that these dependencies have. Similarly, the work we have presented in [4] allows for an analysis of the impact that certain factors have on the performance of service compositions. SLA prediction as discussed in this paper has first been discussed in [14], which is based on some early work of HP Laboratories on SLA monitoring for Web services [15]. In [14], the authors introduced some concepts which are also present in our solution, such as the basic idea of using prediction models based on machine learning techniques, or the trade-off between early prediction and prediction accuracy. However, the authors do not discuss important issues such as the integration of instance and QoS data, or strategies for updating prediction models. Additionally, this work does not take estimates into account, and relatively little technical information about their implementation is publicly available. A second related approach to QoS prediction has been presented recently in [16]. In this paper the focus is on KPI prediction using analysis of event data. Generally, this work exhibits similar limitations as the work described in [14], however, the authors discuss the influence of seasonal cycles on KPIs. This facet has not been examined in our work, even though seasons can arguably be integrated easily in our approach as additional facts.

7 Conclusions

In this paper we have presented an approach to runtime prediction of SLA violations. Central to our approach are checkpoints, which define concrete points in the execution of a composite service at which prediction has to be carried out, facts, which define the input of the prediction, and estimates, which represent predictions about data which is not yet available in the checkpoint. We use techniques from the area of machine learning to construct regression models from recorded historical data to implement predictions in checkpoints. Retraining strategies govern at which times these regression models should be refreshed. Our Java-based implementation uses the WEKA Machine Learning framework to build regression models. Using an illustrative example we have shown that our approach is able to predict SLO values accurately, and does so in near-realtime (with an delay of well below 1 second).

As part of our future work we plan to extend the work presented here in three directions. Firstly, we want to improve the usability of our prototype by improving the GUI, especially with regard to the definition of checkpoints. Currently, this is mostly done on XML code level, which is clearly unsuitable for the targeted business users. Instead, we plan to incorporate a template-based approach, where facts and estimates are as far as possible generated automatically. Secondly, we want to generalize the ideas presented in this paper so that they are also applicable to aggregated SLOs, such as “Average Response Time Per Day”. Thirdly, we plan to extend our prototype to not only report possible SLA violations to a human user, but to actively try to prevent them. This can be done by triggering adaptations in the service compositions, for instance using BPEL’n’Aspects [17]. However, more research needs to be conducted in order to define models of how possible SLA violations can best be linked to adaptation actions, i.e., how to best define which adaptations are best suited to prevent which violations.

Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube).

References

1. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer **11** (2007)

2. Menascé, D.A.: Qos issues in web services. IEEE Internet Computing **6**(6) (2002) 72–75
3. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. 2 edn. Morgan Kaufmann (2005)
4. Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Leymann, F., Dustdar, S.: Monitoring and Analyzing Influential Factors of Business Process Performance. In: EDOC'09: Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference. (2009)
5. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection . In: MW4SOC 2009: Proceedings of the 4rd International Workshop on Middleware for Service Oriented Computing. (2009)
6. Wetzstein, B., Strauch, S., Leymann, F.: Measuring Performance Metrics of WS-BPEL Service Compositions. In: ICNS'09: Proceedings of the Fifth International Conference on Networking and Services, IEEE Computer Society (2009)
7. Haykin, S.: Neural Networks and Learning Machines: A Comprehensive Foundation. 3 edn. Prentice Hall (2008)
8. Quinlan, J.R.: Improved Use of Continuous Attributes in C4.5. Journal of Artificial Intelligence Research **4** (1996) 77–90
9. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo. Technical report, TUV-1841-2009-03, Vienna University of Technology (2009)
10. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web Services on Demand: WSLA-Driven Automated Management. IBM Systems Journal **43**(1) (2004) 136–158
11. Tosic, V., Pagurek, B., Patel, K., Esfandiari, B., Ma, W.: Management applications of the web service offerings language (wsol). Information Systems **30**(7) (2005) 564–586
12. Tosic, V., Ma, W., Pagurek, B., Esfandiari, B.: Web Service Offerings Infrastructure (WSOI) – A Management Infrastructure for XML Web Services. In: NOMS'04: Proceedings of the IEEE/IFIP Network Operations and Management Symposium. (2004) 817–830
13. Bodenstaff, L., Wombacher, A., Reichert, M., Jaeger, M.C.: Monitoring Dependencies for SLAs: The MoDe4SLA Approach. In: SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing, Washington, DC, USA, IEEE Computer Society (2008) 21–29
14. Castellanos, M., Casati, F., Dayal, U., Shan, M.C.: Intelligent Management of SLAs for Composite Web Services. In: DNIS 2003: Proceedings of the 3rd International Workshop on Databases in Networked Information Systems. (2003) 28–41
15. Sahai, A., Machiraju, V., Sayal, M., Moorsel, A.P.A.v., Casati, F.: Automated SLA Monitoring for Web Services. In: DSOM '02: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, London, UK, Springer-Verlag (2002) 28–41
16. Zeng, L., Lingenfelder, C., Lei, H., Chang, H.: Event-Driven Quality of Service Prediction. In: ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing, Berlin, Heidelberg, Springer-Verlag (2008) 147–161
17. Karastoyanova, D., Leymann, F.: BPEL'n'Aspects: Adapting Service Orchestration Logic. In: ICWS 2009: Proceedings of 7th International Conference on Web Services, Los Angeles, CA, USA, IEEE (2009)

Adaptation of Service-Based Systems based on Requirements Engineering and Online Testing

Andreas Gehlert[1], Andreas Metzger[1], Dimka Karastoyanova[2], Raman Kazhamiakin[3], Klaus Pohl[1], Frank Leymann[2], and Marco Pistore[3]

[1] University of Duisburg Essen, Software Systems Engineering, Schtzenbahn 70, 45127 Essen, Germany,
{andreas.gehlert | andreas.metzger | klaus.pohl}@sse.uni-due.de
[2] IAAS, University of Stuttgart, Universitaetsstr. 38, 70569 Stuttgart, Germany,
{dimka.karastoyanova | frank.leymann}@iaas.uni-stuttgart.de
[3] FBK-IRST, via Sommarive 18, 38100 Trento, Italy,
{raman.kazhamiakin | marco.pistore}@fbk.eu

Abstract. Service-orientation offers high flexibility in composing applications from individual services. This flexibility allows dynamically adapting service-based applications during run-time. Various goals for such dynamic adaptations can be addressed, which include (1) aiming to better achieve the users' requirements (perfective adaptation), and (2) repairing and preventing failures (corrective adaptation).

When building adaptive applications that address two or more such goals, precautions must be taken to ensure that the interplay and the interactions between the different types of adaptations are considered. For instance, it is important to understand how the need for the corrective adaptation of the SBA must to be aligned and synchronized with the opportunity for the perfective adaptation of the SBA, as otherwise this can lead to conflicting adaptations. For instance, the corrective part could aim at replacing a failed service A for a service B, while at the same time the perfective part could aim at replacing service A with a service C.

This paper introduces a framework to integrate and align perfective and corrective adaptations, while addressing the problems raised by the interactions between those two kinds of adaptation. The framework is based on two techniques for performing perfective and corrective adaptations respectively. The perfective adaptation technique is based on requirements engineering activities to identify new or improved services. The corrective technique is based on online testing, which enables the prediction of potential future failures of the SBA.

Based on the above techniques, this chapter investigates the interplay and interaction between the two types of adaptation. We demonstrate how perfective and corrective techniques can be integrated in a meaningful way to support the overall adaptation requirements of the service-based application, while avoiding the above problems. As a solution, we propose exploiting an enterprise service registry, which restricts the ways in which a SBA can be adapted. The integrated approach as well as its building blocks are demonstrated through an application scenario from the telecommunication domain.

1 Introduction and Overview

1.1 Motivation and Problem Statement

Organizations increasingly rely on the flexibility offered by service-based applications (SBAs). This flexibility allows those applications to operate in a highly dynamic world, in which the level and quality of service provisioning, (legal) regulations, as well as requirements keep changing and evolving. To respond to those changes, service-based applications need to modify their functionality and quality dynamically depending on the usage situation, context, and deployment platform. In addition those applications will need to react to failures of the constituent services to ensure that they maintain their expected functionality and quality. In such a dynamic setting, evolution and adaptation methods and tools become key to enable those applications to respond to changing conditions. Following the terminology defined by the S-Cube Network of Excellence [33], we refer to evolution as the more traditional modification of a system's requirements, specification, models and so forth during design time ("maintenance"), while we understand adaptation to refer to the modification of a specific instance of a system during runtime. In this chapter, we focus on adaptation of service-based applications and thus address *adaptive service-based applications*.

The adaptation of an SBA can address various goals, such as (1) correcting faults contained in the SBA (corrective adaptation [24, p. 43]), and (2) adapting the SBA to new and yet unknown requirements (perfective adaptation) [45, p. 493].

When building adaptive SBAs that address two or more such goals, precautions must be taken to ensure that the interplay and the interactions between the different types of adaptations are considered. In fact, coordinating those goals is considered one of the significant challenges in self-adaptive software [41]. As an example, it is important to understand how the need for the corrective adaptation of the SBA must be aligned and synchronized with the opportunity for the perfective adaptation of the SBA. Otherwise, this can lead to conflicting adaptations, which need to be avoided. As an example, the corrective part could aim at replacing a failed service A for a service B, while at the same time the perfective part could aim at replacing service A with a service C.

1.2 Solution Idea

To demonstrate how conflicts between adaptation goals can be avoided, we focus on the two adaptation goals introduced above: corrective and perfective adaptation. More specifically, we exploit the following techniques to determine the demand for an adaptation of the SBA (i. e., to determine an adaptation trigger [41]):

Corrective adaptation based on online testing: Online tests of services are performed during runtime (operation) of the SBA to determine possible failures of the SBA's constituent services. A failure of such a test constitutes a corrective adaptation trigger, which could possibly be satisfied by replacing the failed service with an alternative service.

Perfective adaptation based on RE: Typically, enterprises have contract relationships with other business partners. This fact is reflected in the set of services that may be used in SBAs. These partner services usually meet the requirements specified by the requirements engineers in the enterprise. In some cases however, due to the dynamic nature of the service market, new relationships are established with other (previously unknown) partners. If the newly introduced service is better and/or more appropriate (e. g. cheaper or faster), the requirements engineer could recommend the use of this new service and thereby issue a perfective adaptation trigger.

Both of the above techniques share the characteristic that they are proactive in nature, i. e., both techniques lead to "predictive" adaptation triggers. In the case of online testing, the failure of a service could point to a problem of the SBA (which involves this service) in the future. In the case of recommendations from RE, this provides the possibility to improve the SBAs and to anticipate future requirements. Thereby, both of those adaptation drivers, which are the core building blocks of our proposed solution, share a fundamental commonality. This simplifies addressing the problem of synchronizing the two adaptation goals.

In addition, to exploiting this commonality, a further key idea of our approach is to use a central enterprise service registry. This registry contains references to in-house services, e. g. those services provided by the enterprise itself, and to external services, e. g. those services, that are provided by external service providers. Only these services are allowed to be used in the enterprise's service-based applications. Every reference to one of the service is accompanied by a service description (cf. Section 4 for more details). Since the enterprise service registry is a private registry, it can be administrated solely by the enterprise's administrators. On the one hand this enterprise service registry constraints possible adaptation and, therefore, reduces the flexibility of the SBA and, on the other hand, allows us to use techniques (e. g. testing techniques), which require a certain level of stability (cf. Figure 1).

1.3 Focus and Assumptions

In order to focus our chapter, we restrict ourselves to service compositions, which use the workflow-approach, i. e. we assume that a service composition is described in terms of a control flow (sequence of tasks), data flow (data exchange between tasks), exception handling and services, which realise the functionality of the tasks. Therefore, service composition descriptions are organized in two dimensions—control logic and functionality.

To further constrain the scope of the chapter, we consider the exchange of individual services as the only mechanism for performing adaptations of SBAs. This means that we only adapt SBAs by changing the bindings of services (or end points) to the workflow. This means, the modification of the control or data flow structure is not addressed here and will be part of the future work in the S-Cube project.

Finally, we assume that a SBA is a service composition (workflow) and use both terms synonymously in this chapter.

1.4 Outline of the Approach

Figure 1 provides an overview of our approach. Below, we briefly illustrate how the two adaptation goals—perfective and corrective adaptation—are addressed in this book chapter, and how we envision to avoid conflicting adaptations. The remainder of this chapter will provide more details on the individual techniques and their synchronisation:

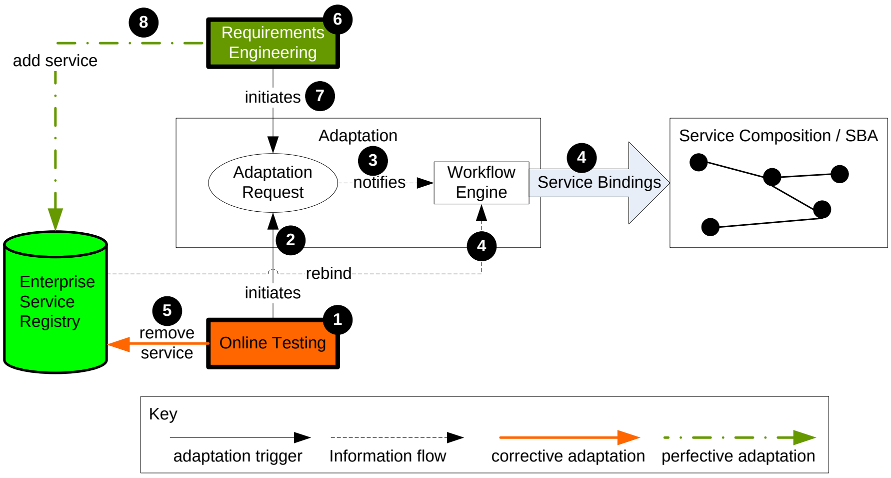


Fig. 1. Overall Approach

Situation 1 – Corrective Adaptation: Assume that online testing finds a fault in one of the SBA's services, which can lead to an overall system failure in the future (① in Figure 1). In this case the online testing activity initiates triggers an adpatation to exchange this service (②). Based on this adaptation trigger, the actual adaptation component of the SBA (③) needs to find an alternative service (e. g., by searching the service registry) and has to notify the workflow engine to bind this alternative service to the service composition in place of the failed service (④). To avoid using the failed service in other SBAs of the enterprise, the online testing activity removes it from the enterprise service registry (⑤).

Situation 2 – Perfective Adaptation: Assume that a new service was discovered during requirements engineering and that this service is cheaper than the previously used one (⑥). If the requirements engineer decides to use this service in the SBA, s/he needs to add it to the enterprise service registry (⑧) and to trigger an adaptation (⑦). This adaptation trigger will eventually notify the workflow engine (③) and similar to situation 1 the workflow engine rebinds the new service to the service composition instead of the old one (④).

For the purpose of this chapter, we assume that an SBA is already running and, therefore, uses services from the enterprise service registry.

During this run-time phase, the online testing activity continuously tests the SBA's services for failures and if a failure is observed, the respective service is removed from the enterprise service registry. The requirements engineering activity continuously searches for new and innovative services and if such a service is found, it is added to the enterprise service registry. Since our approach requires that every service-based application in the enterprise uses services from the enterprise-service registry, this registry together with the workflow adaptability serve as tools to synchronize the activities of the requirements engineer and the online tester. For instance, it is not possible in our scenario that the online tester and the requirements engineer concurrently replace services in the workflow. Since there is only the possiblity to add or remove services to/from the registry the state of the workflow cannot be inconsistent (cf. Section 7.3 for a detailled discussion).

1.5 Structure of Chapter

Following the structure of Figure 1, the remainder of this chapter is organized as follows: In Section 2 we present the relevant state of the art on requirements engineering, monitoring, adaptation and testing on which our research is built. In Section 3 we introduce the scenario, which we use in the text to illustrate our results. Subsequently we describe the requirements engineering (Section 4) and online testing techniques (Section 5), which may trigger adaptation. In Section 6 we explain how the two techniques interact with each other. This integration is then demonstrated with the help of an example in Section 6.2. Section 7 contains a critical discussion of our results. The conclusions are summarized in Section 8.

2 Related Work

The state of the art discussion in this section is structured as follows: In Section 2.1 we first discuss related work that addresses the problem of synchronizing adaptations in the presence of more than one adaptation goal. Section 2.2 describes the related work on requirements engineering, which is relevant for Section 4 of this chapter. Sections 2.3 and 2.4 summarize the related work on monitoring and online testing respectively, which is relevant for Section 5 of this chapter.

2.1 Related Work on Multi-Goal Adaptation

In [41] Salehie and Tahvildari stress that "coordinating [...] goals at different levels of granularity is one of the significant challenges in self-adaptive software." As a result of the literature survey carried out in that paper, the authors reach the conclusion that only very few approaches address more than one goal of adaptation (or, self-* property). In addition, they observe that most approaches that address more than one goal do not systematically coordinate those goals. One concrete, architecture-based approach that addresses multiple goals is introduced by Cheng et al. in [13]. However, rather than addressing high-level goals, such as perfective and corrective adaptation that are addressed in our approach, the authors address conflicting situations between more fine-grained objectives, such as performance and other quality of service characteristics.

2.2 Related Work on Requirements Engineering

Although Tropos was applied in the service domain, these applications do not explain when to adapt a SBA. Aiello and Giorgini for instance explore quality of service aspects using Tropos actor models [2]. The authors use Tropos' formal reasoning techniques in [18] to calculate the fulfilment of a goal structure according to a given set of services. As the approach by Aiello and Giorgini does not cover the adaptation of a SBA, our approach is an extension to [2]. In another approach Penserini et al. explore how Tropos can be used to develop SBAs. However, the authors do not focus on adaptation. Another application of Tropos was put forward by Pistore et al. in [36]. The authors explain how SBAs can be developed by step-wise refining plans and complementing these plans with a formal workflow definition. Since the focus of Pistore et al. is on deriving service compositions, the authors do not cover adaptation issues. The introduction to Tropos in [11] also contains a comparison of goal models to chose the architecture of the software system [11, p. 373]. This comparison is limited only to choosing so called architectural styles and, thus, does not explain adaptation.

A similar approach to ours was put forward by Herold et al. in [20]. The authors related existing components to goal models. This relation is established by so called generic architectural drivers. These drivers enable the selection of existing components, which fit with the goals and soft-goals of the goal model. Herold et al.'s approach focus on finding appropriate components and refining the initial goal model with the help of these components. However, the approach does not address adaptation. Another RE approach, which is similar to ours, was put forward in the Service Centric Systems Engineering (SeCSE) project [27]. In SeCSE initial requirements are formulated as goal models [27, pp. 21] or use cases [22,50,51,52], which are than translated into services queries [27, p. 31]. These services queries are sent to a registry. The resulting services are used to refine the initial set of requirements. However, in SeCSE the focus was on changing the requirements according to the current service provision but not on adapting existing SBAs.

2.3 Related Work on Monitoring for Adaptation

In order to detect events and situations that necessitate an adaptation of a service-based application, the majority of adaptation approaches from the service-oriented computing field resorts to exploiting *monitoring* techniques. Monitoring provides a way to collect and report relevant information about the execution and evolution of a service-based application. Depending on the goal of a particular adaptation approach, different kinds of events are monitored and different techniques are used for this purpose.

In many approaches (e.g., [6,7,17,31]) the events that trigger the adaptation are failures. These failures include typical problems such as application exceptions, network problems and service unavailability [6,31], as well as the violation of expected properties and requirements. In the former case fault monitoring is provided by the underlying platform, while in the latter case specific facilities and tools are necessary. In [7] Baresi et al. define the expected properties in the form of WS-CoL assertions (pre-condiations, post-conditions, and invariants), which define constraints on the functional and quality of service (QoS) parameters of the service composition and its context. In [43] Spanoudakis et al. use properties in the form of complex behavioral requirements expressed in event calculus. In [17] Erradi at al. express expected properties as policies on the QoS parameters in the form of event-condition-action (ECA) rules. When a deviation from the expected QoS parameters is detected, the adaptation is initiated and the application is modified. In such a case, adaptation actions may include re-execution of a particular activity or a fragment of a composition, binding/replacement of a service, applying an alternative process, as well as re-discovering and re-composing services. In [42] Siljee et al. use monitoring to track and collect the information regarding a set of predefined QoS parameters (response time, failure rates, availability) infrastructure characteristics (load, bandwidth) and even context. The collected information is checked against expected values defined as functions of the above parameters, and in case of a deviation, the reconfiguration of the application is triggered.

Summarizing, all these works follow the reactive approach to adaptation, i.e., the modification of the application takes place *after* the critical event happened or a problem occurred.

The situation with reactive adaptation is even more critical for approaches that rely on post-mortem analysis of the application execution. A typical monitoring tool used in such approaches is the analysis of workflow logs [1,19,32]. Using the information about histories of application executions, it is possible to identify problems and non-optimalities of the current business process model and to find ways for improvement by adapting the service-based application. However, once this adaptation happens, many workflow instances might have already been executed in a “wrong” mode.

2.4 Related Work on Online Testing and Regression Testing

The goal of testing is to systematically execute services or service-based applications (service compositions) in order to uncover failures, i.e., deviations of the actual functionality or quality of service from the expected one.

Existing approaches for testing service-based applications mostly focus on testing during design time, which is similar to testing of traditional software systems. There are a few approaches that point to the importance of online testing of service-based applications. In [47] Wang et al. stress the importance of online testing of web-based applications. The authors, furthermore, see monitoring information as a basis for online testing. Deussen et al. propose an online validation platform with an online testing component [14]. In [12] metamorphic online testing is proposed by Chan et al., which uses oracles created during offline testing for online testing. Bai et al. propose adaptive testing in [3,5], where tests are executed during the operation of the service-based application and can be adapted to changes of the application’s environment or of the application itself. Finally, the role of monitoring and testing for validating service-based applications is examined in [10], where the authors propose to use both strategies in combination. However, all these approaches do not exploit testing results for (self-)adaptation.

An approach related to online testing is regression testing. Regression testing aims at checking whether changes of (parts of) a system negatively affect the existing functionality of that system. The typical process is to re-run previously executed test cases. Ruth et al. [39,40] as well as Di Penta et al. [15] propose regression test techniques for Web services. However, none of the techniques addresses how to use test results for the adaptation of service-based applications.

Summarizing, in spite of a number of approaches for online testing and regression testing, none of these approaches targets the problem of proactive adaptation.

An Initial Proposal for Data-Aware Resource Analysis of Orchestrations with Applications to Proactive Monitoring[⋆]

Dragan Ivanović[1], Manuel Carro[1], and Manuel Hermenegildo[1,2]

[1] School of Computer Science, T. University of Madrid (UPM)
[2] IMDEA Software, Spain
idragan@clip.dia.fi.upm.es, {mcarro, herme}@fi.upm.es

Abstract. Several activities in service oriented computing, such as monitoring, automatic composition, and adaptation, can benefit from knowing ahead of time future properties of a given service composition. In this paper we focus on how statically inferred cost functions on input data, which represent safe upper and lower bounds for different cost measures, can be used to predict some runtime QoS-related values (to, e.g., validate compositions at design time) and to compare actual and predicted resource usage at run-time in order to take adaptive actions if needed. In our approach a BPEL-like orchestration is expressed in an intermediate language which is in turn automatically translated into a logic program. Cost and resource analysis tools are applied to infer functions which, depending on the contents of some initial incoming message, return safe upper and lower bounds of some resource usage measure.

Keywords: Service Orchestrations, Resource Analysis, Data-Awareness, Monitoring

1 Introduction

Service Oriented Computing (SOC) [1] is a well-established paradigm which aims at expressing and exploiting the computation possibilities of remotely interacting loosely coupled systems that expose themselves using service interfaces whose description may include operation signatures, behavioral descriptions, security policies, and other features, while the implementation is completely hidden. Several service interfaces can be *put together* to accomplish more complex tasks through the so-called *service compositions*. Such composition is usually written using a general-purpose programming language or some language specifically designed to express SOC compositions [2–4]. Service compositions, in turn, can expose themselves as full-fledged services.

[⋆] The research leading to these results has received funding from the European Community's Seventh Framework Programme under the Network of Excellence S-Cube - Grant Agreement n° 215483. Manuel Carro and Manuel Hermenegildo were also partially supported by Spanish MEC project 2008-05624/TIN *DOVES* and project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo was also partially supported by EU projects FET IST-15905 *MOBIUS*, FET IST-231620 *HATS*, and 06042-ESPASS.

One key distinguishing feature of SOC systems is that they are expected to live and be active during long periods of time and span across geographical and administrative boundaries. This makes it necessary to include monitoring and adaptation capabilities at the heart of SOC. Monitoring checks the actual behavior of the system and compares it with the expected one. If deviations are too large, an adaptation (which may involve, e.g., rebinding to different services with compatible semantics and better behavior) may become necessary. When deviations are predicted ahead of time instead of detected when they happen, the system is performing proactive monitoring. This is, of course, more complex but also more interesting and useful, as it performs *prevention* instead of *healing*.

Monitoring usually requires *snooping* the actually delivered quality of service (QoS) in order to detect (undesired) underperformance with respect to the planned execution. However, comparing actual and expected QoS of a composition —even assuming the composition does not change over time— is far from trivial. Clearly, the more accurately one can calculate the expected QoS, the better predictions can be made. In estimating QoS behavior, two factors, at least, have to be considered:

- The structure of the composition itself, i.e., what it does with incoming requests and which other services it invokes and how (which is, initially, under the control of the designer or, at least, completely known at any moment in time), and
- The variations on the environment, such as network links going down or external services not meeting the expected deadlines, which are contingent and usually out of control.

As an example, when predicting the total time spent in sending and receiving messages one must take into account, on one hand, the number of service invocations in each direction (which depends on the structure of the composition) and, on the other hand, the time sending and receiving every message takes, which is outside the control of the composition.

Of these two sources of information, the latter has been extensively studied [5–8], while the former has been, to our knowledge, less deeply explored. In particular, certain characteristics of some SOC-oriented languages, like fault handling, different patterns for message-based invocation, etc. have not been appropriately taken into account: often, problematic constructs of the language under study were ignored. Also, information such as the actual data received through a service invocation has been recognized as relevant [9, 10] but has not been correctly addressed so far. As we will see in Section 2, the actual message contents can greatly influence the runtime behavior of a composition (e.g., reserving hotels for one person is, from the point of view of spent resources, not the same as reserving for one hundred, since more messages are sent, more bandwidth is spent, etc.), which makes prediction techniques that do not take run-time parameters into account potentially inaccurate.

In this paper we will focus on developing a methodology, based on previous experience on automatic complexity analysis [11–13], which can generate correct approximations of cost functions measuring a variety of relevant execution characteristics via translation to an intermediate language (Sections 3, 4, and 5). These functions use (abstractions of) incoming messages in order to derive correct upper and lower

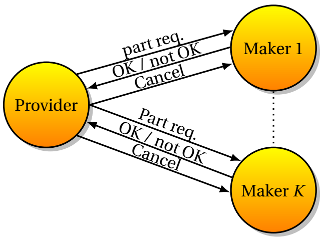


Fig. 1. Simplified hotel reservation system.

bounds which depend on the input data and which are potentially more accurate that data-unaware approximations. In Section 6 we show how these functions can be used to help monitoring make better decisions.

We want to note that correct data-aware cost functions can in general be applied to any situation where a more informed QoS estimation is an advantage. In particular, QoS-driven service composition [14–16] can use them in order to select better service providers given information on which kind of requests are expected. In a related setting, adaptation mechanisms can also benefit from such a knowledge [17].[3]

2 A Motivating Example

We illustrate with a simple example how actual data can be taken into account when generating QoS expressions for service compositions.

Example 1 *Figure 1 shows a simple hotel reservation system. The Client (e.g., a browser maybe operated by a final user or by a travel agency) gets in touch with a Booking Agency and requests N hotel rooms. The Booking Agency runs (or accesses) a composite service which tries a number K of hotels until it either finds all N rooms, or replies with a* no rooms available *message. Moreover, the service books rooms one person at a time as they are available, and, if after scanning all the hotels, not enough rooms are available, it revokes the reservations made so far by means of cancellation messages. A hotel that reports that it has no more available rooms is excluded from further search. We assume that one message is used to for each room query, one for each confirm / reject reply, and one for each reservation revocation.*

Note that it is unlikely that the whole process can be made as a single transaction because the reservation system of the different hotels may very well be disconnected; therefore it has to be instrumented at the level of composition.

We will assume that we are interested in the number of messages sent / received. There are several reasons for this: in a real system, message exchange can carry a sizable overhead, thus significantly affecting the actual execution time; it is possible that hotel reservation services take a toll on every message they answer; and the Booking Service could also charge some amount of money per message.

[3] Our related work on applying the derived cost functions to guide adaptation [17] involves (re)binding of service candidates on that basis. Although the technique for deriving cost bound functions is the same, we here focus on the different problem of application to proactive monitoring.

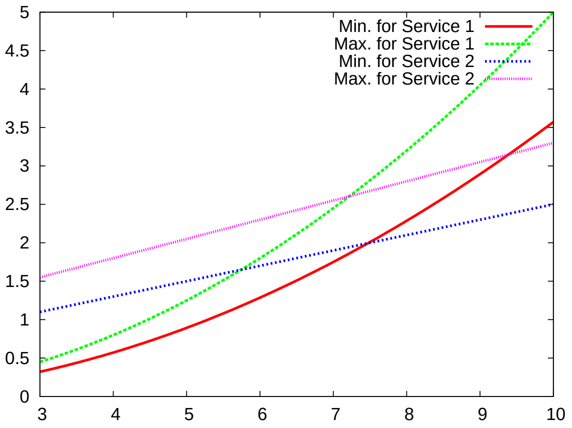


Fig. 2. Upper and lower bounds for two services.

Assuming $K \geq N$, the *minimum* number of messages that can be sent before returning is $2N$, corresponding to a successful reservation (N successful requests and replies to the same hotel) while the *maximum* number of messages is $2K + 3(N - 1)$, corresponding to the worst case unsuccessful reservation ($N - 1$ successful reservations, plus one last unsuccessful reservation which triggers cancellation of the $N - 1$ successful reservations). Between these extremes, the maximum for a successful reservation would be $2K + 2(N - 1)$ messages.

The analysis is not trivial, even for this very simple case, and depends, on one hand, on the internal logic of the composition and on the other hand on the values of N and K, which should be considered parameters for the composition, since it is more likely that the hotels are listed in a separate registry, than hardwired into the composition code.

Compared with probabilistic approximations, the following differences can be pointed out:

- In the dataless formulation, the impact of loops and conditionals can at most be estimated based on, for example, historical data. It cannot be used to actually give any *guarantee*, as the value for any QoS characteristic will be constant regardless of the actual input values for K and N.
- Additionally, safe upper and lower approximations (e.g., bounds) cannot be usually obtained, as these probabilistic formulae only use a single number representing some type of average.
- In the case of QoS-aware matchmaking or rebinding, comparing two different service compositions ignores the functional dependency that the QoS has on the data. Figure 2 portrays the upper and lower bounds of two compositions for

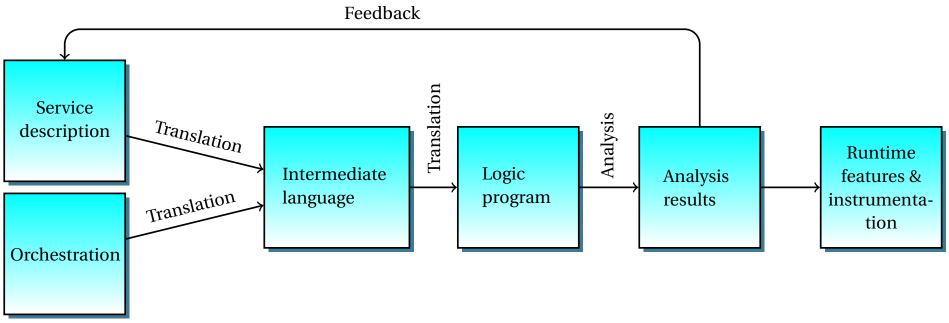


Fig. 3. The overall process.

> some QoS as a function of a single input parameter. For same ranges of data input one composition is preferable over the other, while in the central, *shared* zone the information we have is not enough to decide. Knowing in which area of the graph we are located is relevant in order to make the best matching choice.

We primarily aim at inferring functions counting a number of relevant *events*.[4] To this end, we follow the approach to resource-oriented analysis of [18, 19]. The fundamental idea is to specify how much every part of a composition contributes to the usage of some resource, and derive cost functions based on that specification. A key aspect is that, for operations which we cannot analyze (because they are external to the composition process, such as database accesses) but whose cost we want to take into account, the specifications of how much resources they consume can be made through functions that take into account the actual data.

3 An Overall Description of the Analysis Process

Figure 3 shows the overall picture of the process we present. The orchestration description (which can be written e.g. in BPEL, although our approach should be valid for other orchestration languages), together with meta-information usually contained in the related WSDL document, is translated into an intermediate language whose constructs are shown in Table 1.

The declarations in Table 1 describe namespace prefixes (used for qualified names), XML-schema-derived data types for messages, and service port types. Besides, the intermediate language allows declaring external services that are not analyzed, but have some trusted properties that are either results of a separate analysis or *a priori* assumptions.

A BPEL process definition is translated into a service definition which associates a port name and an operation with a BPEL-style activity that represents the orchestration body. The choice of activities in Table 1 is driven by the key features of the

[4] Note that the technique we are building on was primarily applied to compute execution steps, which are close to execution time.

Declarations and definitions	
Namespace prefix declaration	`:- prefix(` *Prefix*, *NamespaceURI*`).`
Message or complex type definition	`:- struct(` *QName*, *Members*`).`
Port type definition	`:- port_type(` *QName*, *Operations*`).`
External service declaration	`:- service(` *PortName*, *Operation*, `{` *Trusted properties* `}).`

Service definition	`service(` *Port*, *Operation*, *InMsg*`[,` *OutMsg*`])`
	`:-` *Activity* `.`

Activities	
Do nothing	`empty`
Assignment to variable / part	*VarExpr* `<-` *Expr*
Service invocation	`invoke(` *PortName*, *Operation*, *OutMsg*, *InMsg*`)`
Terminating with a response	`reply(` *OutMsg*`)`
Sequence	*Activity*$_1$, *Activity*$_2$
Conditional execution	`if(` *Cond*, *Activity*$_1$, *Activity*$_2$`)`
While loop	`while(` *Cond*, *Activity*`)`
Repeat-until loop	`repeatUntil(` *Activity*, *Cond*`)`
For-each loop	`forEach(` *Counter*, *Start*, *End*, *Activity*`)`
Scope	`scope(` *VarDeclarations*, *Activities and Handlers*`)`

Scope fault handler	`handler(` *Activity*`)`
	`handler(` *FaultName*, *Activity*`)`

Parallel flow with dependencies	`flow(` *LinkDeclarations*, *Activities*`)`
Dependent activity in a flow	`float(` *Attributes*, *Activity*`)`

Table 1. Elements of an abstract description of an orchestration in the intermediate language.

subset of BPEL we are concerned with. In particular, we restrict ourselves to orchestrations that accept a single input message and terminate their work by either dispatching a reply or failing. That is by far the most common type of orchestration, although extension to orchestrations that may accept several different input messages is straightforward using the native non-determinism available in the target platform (see below). Support for resource analysis of stateful service callbacks is a subject for future work.

Next, the intermediate representation is translated into a logic programming language (Ciao [20]) augmented with assertions [21] which allow expressing types and modes (i.e., which arguments are input or output) as well as resource definitions and functions describing resource consumption bounds. The type and mode assertions help the analyzer to "understand" more precisely what the original program meant —i.e., not to lose the information about data directionality that was present in the original orchestration. Intuitively, the reason to do this is that since Prolog has a very free view of types and a complex control strategy (including built-in backtracking), a naïve, unannotated translation would generate a program exhibiting more possible behaviors than those of the original BPEL program, and therefore the analysis results would very likely lose precision. The logic program resulting from the translation is fed to the resource consumption analyzer of the Ciao preprocessor (CiaoPP), which is able to infer upper and lower bounds for the generalized cost / complexity of a logic program [11, 12, 18, 19].

The results of the analysis (the cost functions) are fed back to service description, thus adding more information on the service, which can be stored in a registry and used by cost-sensitive binding and matchmaking algorithms. The results are also used to inform the infrastructure and monitoring parts of the SOC architecture on the expected runtime features of the orchestration and thus help deployment, compilation into object code, and run-time instrumentation.

An important observation regarding the translation is that, in general, we do not need the generated logic program to be strictly faithful to the operational semantics of BPEL: it has to reflect just the necessary part of the semantics that will ensure that the analyzers will infer correct information (i.e., safe approximations), with minimal precision loss due to the translation. However, in our case the translated program *is* executable (although not operationally equivalent to the BPEL process) and mirrors closely the operational semantics of the BPEL process under analysis.

4 An Outline of the Translation from BPEL to Logic Programs

In this section we will briefly describe the translation of BPEL process definitions, via the intermediate language, to a logic program that is analyzed by existing tools. A set of BPEL processes which form a (small) service network are taken as the input to the process and the result is a single file with a logic program, where BPEL processes are mapped onto predicates which call each other when the original BPEL processes would invoke another service. In order for the final code to be amenable to analysis, we currently restrict ourselves to a subset of BPEL, which notwithstanding we consider rich enough to express an ample class of interesting real-life cases.

4.1 Restrictions on Input Orchestrations and Correspondence with BPEL

We restrict our analysis to orchestrations that follow a *receive–reply* interaction pattern, where processing activities take place after reception of an initiating message and finish dispatching either a reply or a fault notification. Another behavioral restriction is that we currently do not support analysis of stateful service callbacks using correlation sets or WS-Addressing schemes. In future work, we plan to relax both restrictions by identifying orchestration fragments that correspond to the *receive–reply* pattern, isolating them into sub-processes, and analyzing them in the same way we now treat whole orchestrations.

The activity constructs in the intermediate language in Table 1 are inspired by the key features of BPEL but are applicable to other abstract or executable orchestration languages. Some activity constructs (`empty`, assignment, sequence,...) are commonly found in programming languages. The key constructs for modeling orchestration workflows are `flow`, `float`, `scope/handler`, and `invoke`.

In contrast to the structured workflow patterns expressed by UML activity/sequence diagrams, BPEL's `flow` construct can express a wider class of concurrent workflows, where concurrency and dependencies between activities are expressed by means of precondition formulas involving tri-state logical link variables,

```
:- regtype 'acme->reservationData'/1.
'acme->reservationData'('acme->reservationData'(A, B, C)):-
    num(A), num(B), list(C, 'acme->personInfo').

:- regtype 'acme->personInfo'/1.
'acme->personInfo'('acme->personInfo'(A, B)):-
   atm(A), atm(B).
```

Fig. 4. Translation of types.

with optional dead-path elimination. The `float` construct in the intermediate language annotates an activity within a `flow` with a description of outgoing links and their values, join conditions based on incoming links, and a specification of the behavior in case of a join failure.

The main purpose of `scope` constructs in BPEL is to introduce local variables, fault, and compensation handlers. In our intermediate language, `scope` serves this purpose, with the exception of compensation handlers, which we do not directly support. Compensation handlers in BPEL contain logic that "undoes" effects of a successfully completed scope. As such, a compensation handler is a pseudo-subroutine attached to a scope, which must be explicitly invoked from a fault handler or compensation handler of an enclosing scope. However, the BPEL specification requires compensation handlers to operate on a snapshot of the scope's variables made on successful completion of a scope, including each individual iteration of loop body, which introduces considerable problems for the analysis as it is now. If we ignore value snapshots, BPEL compensation handlers can be inlined at the place of their invocation.

4.2 Type Translation and Data Handling

Services communicate using complex XML data structures whose typing information is given by an XML Schema. The state of an executing orchestration consists of a number of variables that have simple or complex types, including variables that hold inbound and outgoing messages. For the purpose of simplicity, we abstract the multitude of simple types in XML Schemata into just three disjoint types: `numbers`, `atoms`, representing strings, and `booleans`.

WSDL message types and custom complex types from XML Schemata are translated into the intermediate representation and finally into the typing / assertion language of Ciao. These type definitions are used to annotate the translated program and are eventually used by the analyzer. Figure 4 shows an automatically obtained actual translation for the hotel reservation scenario in Example 1. The type name `'acme->reservationData'` is a structure with the same name and with three fields: two numbers and a list of elements of type `'acme->personInfo'`. Each of these elements is in turn a structure with two fields being an atom each.

Following BPEL, we use a subset of XPath as the expression language, which allows node navigation only along the descendant and attribute axes, to ensure that navigation is statically decidable based on structural typing only. The expression `'$req.body/item[1]/@qty'` in the intermediate language refers to the attribute `qty` of the first `item` element in the `body` part of a message stored in variable req. A set of standard XPath operators and basic functions is supported, including position() and last().

To assist the analyzer in tracking component values and correlating the changes made to them, we take the approach of statically decomposing lists and XML structures in an execution environment into their components, and passing them around explicitly as predicate arguments from that point onwards. Unfolded structures no longer need to be passed along with components, since they can be reconstructed on demand. The resulting code is less readable for a human, but more amenable to analysis.[5]

For instance, to access the third element of a list stored as an opaque object in a variable, the list has to be decomposed into head and tail subcomponents, and the process has to be repeated until the third element of the list is reached. From that point, the list can be reconstructed on demand from the first three elements and the remainder, and therefore need not be explicitly passed in predicate calls. However, if the list is assigned (from an expression or by receiving a reply message), we cannot guarantee any more that it has at least three elements, and therefore the list once again becomes an opaque object. The same logic applies to other data structures and their components.

4.3 Basic Service and Activity Translation

The basic idea of the automatic translation from the intermediate language is to keep track of the functional dependency of the resulting response message on the input message with which a service is invoked. Here we present the translation scheme based on generation of clauses of a logic program [17] that can be automatically analyzed for resource usage. An orchestration S is translated into a predicate:

$$s(\bar{x}, y) \leftarrow [\![A]\!]_{\eta_0}(y)$$

where $\bar{x}$ represents components of the input message, y stands for the response message, and $[\![A]\!]_{\eta_0}(y)$ is the translation of the orchestration body A with respect to the initial service environment η_0. An environment maps symbolic (sub)component names (which denote message parts, nested XML elements and attributes, and scalars) to logical terms. Each variable in the environment is either a scalar or a tree-like structure where component nodes branch from structure nodes, up to some depth of unfolding, as explained in the previous subsection. Unfolded structures in an environment (the internal nodes) can always be recursively reconstructed from their components (children nodes). Consequently, the entire environment can be represented by the leaf nodes. When η appears in an argument position, it stands for the list of leaf nodes in η. Leaf nodes of the initial environment η_0 are the list $\bar{x}$ of input message components.

The translation operates on a non-empty sequence of activities, which we can write as $\langle A|C\rangle$, where A is the first activity, and C is the continuation sequence, which

[5] The alternative being writing in Prolog the counterparts for the supported XPath operations and let the analyzers deal directly with them. In our experience, this introduces too much precision loss, and therefore we opted for a more complex translation.

may be empty (ε). We write $[\![A|C]\!]$ to denote the translation of $\langle A|C\rangle$, and, as a shorthand, $[\![A]\!]$ to denote translation of $\langle A|\varepsilon\rangle$. This allows us to normalize translation of a sequence (A_i, A_j) by extending the continuation:

$$[\![(A_i, A_j)|C]\!]_\eta(y) \mapsto [\![A_i|\langle A_j|C\rangle]\!]_\eta(y)\,.$$

Activity `reply(`v`)` terminates the orchestration and sends the reply contained in variable v in the current environment:

$$[\![\mathtt{reply}(v)|C]\!]_\eta(y) \equiv y = \eta(v)\,.$$

Raising a fault with `throw` is translated into a logical failure ($[\![\mathtt{throw}|C]\!]_\eta(y) \equiv$ `fail`), which can be caught on backtracking by fault handlers. The `empty` activity is ignored, so that $[\![\mathtt{empty}|C]\!]_\eta(y) \mapsto [\![C]\!]_\eta(y)$.

For any activity A_i, other than a sequence, `empty`, `reply`, and `throw`, the translation is a predicate call:

$$[\![A_i|C]\!]_\eta(y) \mapsto a_i(\eta, y)\,,$$

where clauses generated for a_i depend on A_i, η, and C. First we look at the case when $A_i \equiv x{<}-e$, i.e., the XPath expression e is evaluated and assigned to the environment element x (a variable or its component). The generated clause has several segments:

$$a_i(\eta, y) \leftarrow [e:E]_\eta, [E/x]_\eta^{\eta'}, [\![C]\!]_{\eta'}(y)\,.$$

where $[e:E]_\eta$ stands for evaluation of e into term E in the environment η, and $[E/x]_\eta^{\eta'}$ stands for mutation of η into η' as the result of assigning E to x. Likewise, in case of an external service invocation, $A_i \equiv \mathtt{invoke}(p, o, v, w)$, the generated clause has the form:

$$a_i(\eta, y) \leftarrow s_{po}(\eta(v), E), [E/w]_\eta^{\eta'}, [\![C]\!]_{\eta'}(y)\,,$$

where s_{po} is the translation of a service implementing operation o on port type p, variable v holds the input message, and variable w receives the reply. For $A_i \equiv \mathtt{if}(c, A_j, A_k)$, two clauses are generated:

$$\begin{aligned} a_i(\eta, y) &\leftarrow [c?]_\eta \quad, [\![A_j|C]\!]_\eta(y) \\ a_i(\eta, y) &\leftarrow [\neg c?]_\eta, [\![A_k|C]\!]_\eta(y) \end{aligned}$$

where $[c?]_\eta$ stands for code that succeeds if and only if the boolean condition c evaluates to `true`. On the basis of `if`, we generate recursive clauses for the case $A_i \equiv \mathtt{while}(c, A_j)$:

$$\begin{aligned} a_i(\eta, y) &\leftarrow [c?]_\eta \quad, [\![A_j|\langle A_i|C\rangle]\!]_{\eta'}(y) \\ a_i(\eta, y) &\leftarrow [\neg c?]_\eta, [\![C]\!]_\eta(y) \end{aligned}$$

Note how reappearance of A_i in the first clause leads to a recursive definition of the translation scheme. The above translation is however not circular, because we already know that $[\![A_i|C]\!]_\eta(y) \equiv a_i(\eta, y)$. Other looping constructs, such as `repeatUntil` and `forEach` reduce to `while`.

4.4 Translation for Scopes and Flows

The translation of scopes involves changing the environment on entry and exit, and has to ensure the execution of a fault handler unless the body scope ends successfully. In $A_i \equiv \texttt{scope}(D, A, H_1, H_2, \ldots, H_N)$, D denotes new variable declarations, A is the body of the scope, and H_i are fault handlers. $N+1$ clauses are generated for a_i, one for A and each of the handlers. Each of the clauses uses cut to prevent execution of subsequent clauses in case that the scope body / handler attached to the clause completes successfully. Since the process itself can be seen as a scope, and it normally needs a variable to hold the output message, in the intermediate language we use an abbreviation:

$$\texttt{service}(p, o, x, y) \leftarrow A$$

for:

$$\texttt{service}(p, o, x) \leftarrow \texttt{scope}([y : \textit{ReplyType}],\ (A,\ \texttt{reply('\$}y\texttt{')})) \,.$$

The translation of a `flow` is done following the usual BPEL semantics [22], but without operationally parallelizing the execution. Instead, we are interested in total resource consumption of a `flow` construct, irrespective of the actual number of available threads. A $\texttt{float}(D, A)$ construct appearing in the body of a `flow` uses attributes D to annotate activity A with input link dependencies and output transition. Links are internally declared as Boolean variables. The floating activities are ordered so that the link dependencies are respected. As in BPEL itself, there can be no circular link dependencies. After reordering, a `flow` effectively translates to a sequence, and each $\texttt{float}(D_j, A_j)$ is transformed into:

$$\texttt{if}(c_j, (A_j, \texttt{'\$}o\texttt{' <- 'true()'}), \Phi)$$

where c_j is a join condition specified in D_j, o is the name of the outgoing link, and Φ covers the case when c evaluates to `false`. When the `suppresJoinFailure` property is disabled, we simply have $\Phi \equiv \texttt{throw(bpel:joinFailure)}$. Otherwise, $\Phi \equiv \texttt{'\$}o\texttt{' <- 'false()'}$.

4.5 Accounting for Unavailable Code

So far we have assumed that the analysis operates on a static composition whose code is available. The same approach can be easily extended to the case where we have a collection of interacting compositions, with statically available code, whether or not these compositions are expected to be deployed locally or remotely. However, there are cases where such code may not be available such as, for example, when some provider does not want to reveal which code is being run on its servers. In such scenarios it is still possible to exploit the partial statically inferred resource usage information to drive cost-sensitive adaptation [17].

Note also that code disclosure concerns may not present a problem for static analysis. The analysis, while starting with an executable (e.g. BPEL) code, does not actually act on such code directly, but rather on some abstraction in the intermediate language which can hide some details. Providers may offer this abstract code in order for third parties to check the complexity claims of the providers. By doing so they

```
:- struct( hotres:resRequest, [
       part( body): struct( hotres:resData)]).

:- struct( hotres:resResponse, [
       part( body): struct( hotres: resData)]).

:- struct( hotres:resData, [
       child( hotres:personCount): number,
       child( hotres:priceLimit): number,
       child( hotres:person):
 list( struct( hotres:persInfo)) ]).

:- struct( hotres:persInfo, [
       attribute( '':firstName): atom,
       attribute( '':lastName): atom,
       child( hotres:hotelName): atom,
       child( hotres:roomNo): number ]).

:- port( hotres:agency, [
       reserveGroup( struct( hotres:resRequest)):
  struct( hotres:resResponse) ]).

:- port( hotres:hotel, [
       reserveSingle( struct( hotres:persInfo)):
                      struct( hotres:persInfo),
       cancelReservation( struct( hotres:persInfo)):
 struct( hotres:persInfo) ]).
```

```
service( hotres:agency, reserveGroup, '$req', '$resp'):-
[
 '$resp.body/hotres:personCount'<-0,
 '$resp.body/hotres:person'<-'$req.body/hotres:person',
 scope( [i:number],
 [   '$i' <- 1,
   while( '$req.body/hotres:personCount>0',
    [
     scope( [p: struct( hotres:persInfo),
     r: struct( hotres:persInfo)],
     [   '$p'<- '$req.body/hotres:person[$i]',
       invoke( hotres:hotel, reserveSingle, '$p', '$r'),
       if( '$r/hotres:roomNo>0',
           '$resp.body/hotres:person[$i]'<-'$r',
           throw( hotres:unableToReserveGroup) ),
       handler(
       [   while( '$i>1',
         [   '$i'<- '$i - 1',
             '$p'<- '$resp.body/hotres:person[$i]',
             invoke( hotres:hotel, cancelReservation,
                     '$p','$r')]),
             throw( hotres:unableToCompleteRequest) ])
     ]),
     '$i' <- '$i+1',
     '$req.body/hotres:personCount' <-
         '$req.body/hotres:personCount - 1' ]) ]) ].
```

Fig. 5. Abstract representation of a group booking process

would increase the confidence of their clients without revealing more than strictly necessary. In other cases, while even this code may not be available, the owner of the service can provide sufficient information in the form of resource assertions which describe the resource consumption behavior without disclosing code in the least.

5 An Example of Translation and Analysis

We will illustrate the process of analysis by using a description of an orchestration, translating it into a logic program, and reasoning on the results of applying to it a resource usage analysis.

We use a representation of a process that performs hotel booking, along the lines (but slightly simplified, for space reasons) of the example used in Section 2. For compactness, we present the abstract description of this orchestration in our internal representation form instead of plain BPEL (Figure 5). This representation contains information that is both found in the WSDL document (data types, interface descriptions) and in the process definition itself (the processing logic).

The orchestration traverses the list of people to book a room for and tries to reserve a room in a hotel by invoking an external hotel service.[6] If that is not possible, or if a failure arises, a failure handler is activated that tries to cancel the reservations that were already made before signaling failure to the client.

The translation of the orchestration produces an annotated logic program, some of whose parts we present in Figure 5. Part (a) shows the translation of the entry point of the service, along with an `entry` annotation that helps the analyzer understand what the input arguments are. The input message is unfolded into the first three arguments (A, B, C), and D plays the role of ω. Part (b) shows the translation of the main `while` loop, and the second clause finishes the process by constructing the

[6] This is a difference from Example 1: the orchestration does not query different hotels.

answer from the current value of the response variable. Part (c) shows the translation of the service invocation, with previous unfolding of the outgoing message, and subsequent pruning of the response variable data tree.

```
:- entry ’service_hotres->agency->reserveGroup’/4
 :{gnd,num}*{gnd,num}*{gnd,’list_of_hotres->persInfo’}*var.

’service_hotres->agency->reserveGroup’(A,B,C,D) :-
 act_1( A, B, C, 0, 0, [], D).
```

(a) Translation of the entry point to the process.

```
act_4( A, B, C, D, E, F, G, H):-
 ----(this is act_4:while(’$req.body/hotres:personCount>0’)),
 A>0, !, act_5( A, B, C, D, E, F, G, H).
act_4( _, _, _, D, E, F, _, ’hotres->resResponse’( D, E, F)).
```

(b) Translation of the main `while` loop.

```
act_7( A, B, C, D, E, F, G, H, _, _, _, _, M):-
 ----(this is act_7:invoke( hotres:hotel, reserveSingle, ’$p’, ’$r’)),
 H=’hotres->persInfo’(N, O, P, Q),
 ’service_hotres->hotel->reserveSingle’( N, O, P, Q, R),
 act_8( A, B, C, D, E, F, G, N, O, P, Q, R, M).
```

(c) Translation of an external service invocation.

Fig. 6. Translation into parts of a logic program.

The resource analysis finds out how many times some specific operations will be called during the execution of the process. The resources we are interested in this example are: the number of all basic activities performed (assignments, external invocations); the number of invocations of individual room reservations (operation `reserveSingle` at the hotel service); and the number of invocations of reservation cancellations (operation `cancelReservation` at the hotel service). From the number of invocations it is easy to deduce the number of messages exchanged during the execution of the process: a single reservation counts as two, and a cancellation counts as one message. The results are displayed in Table 2, where the estimated upper and lower bounds are expressed as a function of the initiating request.

We model processing of a single reservation failure with fault handling that interrupts the normal (nominal) flow and triggers cancellations. We differentiate explicitly between the case with costs of fault processing included, which gives wider, more cautious estimates, and the case in which the execution is successful (i.e., without fault generation and handling). These two cases were obtained by means of different translations which explicitly generated or not Prolog code corresponding to the fault handling. The results in Table 2 correspond to best and worst case estimates from Example 1 . The upper bound on number of messages with fault handling $3N-1$

	With fault handling		**Without fault handling**	
Resource	lower bound	upper bound	lower bound	upper bound
Basic activities	2	$7N$	$5N+2$	$5N+2$
Single reservations	0	N	N	N
Cancellations	0	$N-1$	0	0
No. of messages	0	$3N-1$	$2N$	$2N$

Note: In the above formula, N stands for the value of the input argument `$req.body/hotres:personCount`, taken as a non-negative integer.

Table 2. Resource analysis results for the group reservation service

corresponds to $2K+3(N-1)$ with $K=1$, and the lower bound for the case without fault handling is $2N$.

6 Cost Functions for Monitoring

As briefly discussed in Section 1, the expected value of some QoS characteristics can be derived from the value of some cost functions and the (expected) value of some environment characteristics. In this section we will elaborate on that point and we will sketch how the availability of cost functions can be used to perform proactive monitoring.

6.1 QoS Metrics and Cost Functions

The precise cost function which is needed to express some QoS characteristic depends on the QoS metric itself. For example, if bandwidth consumption is involved in the measure of some QoS, then the number of messages and size of each message is relevant, but the number of executed activities is not directly relevant (although possibly related). However, the cost function by itself cannot in general convey all the information necessary to represent a QoS function: some data which come from the environment is needed. Therefore, and for some QoS metrics, an interval of lower and upper bounds depending on the input data can be expressed as

$$QoS_{\langle L,U\rangle}(n) = \langle cost_L(n) \oplus env_L,\ cost_U(n) \oplus env_U\rangle \tag{1}$$

where the left and right components of the tuple are the expected lower and upper bounds for the quality of service, $cost_X(n)$ is some suitable analytically determined cost / resource consumption function, env_X represents the minimum and maximum influence of the environment conditions on the QoS at hand, and $\oplus$ is an operation which combines together the cost functions and the environment conditions.

For example, in case of the execution time of a single process, $cost_X(n)$ can be the number of activities executed, and env_L and env_U the maximum and minimum time a single activity can take (which depends on the machine executing it, the executing engine, the operating system, etc.), and $\oplus$ would be just multiplication. Since $cost_L(n)$ and $cost_U(n)$ are, respectively, the lower and the upper bound, then if we assume that env_L and env_U are also correct lower and upper bounds, the calculated QoS will be a correct lower and upper bounds of the actual (runtime) QoS values.

Note that this generic scheme can admit variations: for example a more accurate approximation of execution times can be inferred by assigning a different weight to each type of activity. In this case, env_X would actually be an array with a component for the execution time for every type of activity, $cost_X(n)$ would also be an array counting how many times every type of activity is executed, and ⊕ would be the vector dot product.

6.2 QoS and Cost Functions During Composition Execution

Given some QoS characteristic which we assume fixed, Equation (1) relates it to a cost function. It is always the case that the general cost function of a composition is made up of several parts, each one referring to a structural part of the composition. As an example, the upper bound of the cost of an `if-then-else` construct in terms of, for example, executed activities, is the upper bound of the condition plus the maximum of upper bounds of the `then` and the `else` parts.

We can associate to every program point a measure of how much cost / resources remain to be spent in the rest of the execution.[7] For example, in the `if-then-else` example before, once the `if` part is done, what remains is either the `then` or the `else` part. In general this measure depends not only the point in the service composition, but also on the values of the data at that moment: for example, in a loop, where the same activity is executed several times, less "cost" is left until the end of the execution after every iteration, even in the same point of the composition. The difference comes from the different state of the variables, and this is one of the reasons why taking data into account is beneficial.

There is, therefore, a notion of "pending" QoS, which comes from using these composition-point cost functions together with the environment characteristics: for example, from the activities remaining to be executed and the expected time of every activity, the remaining time to the completion of the service activation would be a "pending" QoS. This is, of course, interesting from the monitoring point of view. Assuming that a composition has been designed and approved on the basis of the expected lower and upper bounds of QoS (i.e., the required QoS adequately falls within these bounds), then deviations of the environmental characteristics can be used to predict more accurately what will be the QoS at some future point by dynamically combining the cost / resource consumption functions (which we are assuming do not change during service execution) with the actual environment conditions, which may deviate from those initially assumed.

Figure 7 exemplifies such a situation. Let us assume we are interested on some QoS metric of a composition, whose value must not go over some limit *Max*. Therefore, we should use cost functions and environment characteristics representing safe upper bounds: if the upper bound is smaller than some limit, then we have the guar-

[7] In fact the initial cost is just a measure of how much it remains to be spend assuming that, since the execution has not started, a total of 0 has been spent.

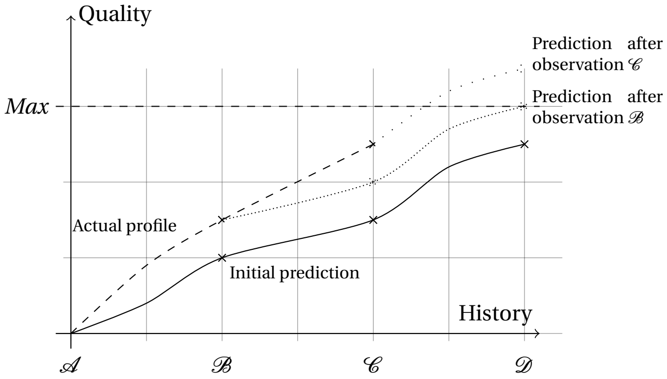


Fig. 7. Actual and predicted QoS throughout history.

antee we need.[8] Symmetrically, if we are concerned with a QoS attribute whose value should not go below some minimum, we would use lower bounds instead. We designate four points ($\mathscr{A}$, $\mathscr{B}$, $\mathscr{C}$, and $\mathscr{D}$) in the execution of some composition and we will focus on how monitoring at these points can be done proactively with the use of cost functions. In Figure 7 the solid line represents the initial running QoS predicted taking into account the statically inferred cost functions and the expected environment conditions. The dashed line represents the actual observed QoS.

At point $\mathscr{B}$, the actual quality has deviated with respect to the predicted one. Since the composition has not changed, and thus neither have the cost functions, we can conclude that the deviation can only be due to a change in the environment behavior (e.g., additional load on a server or a faulty network). A new prediction for the future can be done by using the observed influence of the environment so far and the existing cost function. This new prediction curve (densely dotted) still ends, at point $\mathscr{D}$, within the limits of the acceptable range *Max*. However, at point $\mathscr{C}$ a new observation gives yet higher values for the QoS value and, therefore, for the influence of the environment. Yet another function and associated plot curve (sparsely dotted) can be constructed which predicts that it there is the possibility that the execution finishes violating the QoS constraints. Therefore, at point $\mathscr{C}$ we can raise an alarm and maybe trigger an adaptation procedure. Note that we in fact have detected a problem before it actually appeared. In order for this technique to work in complex service compositions with loops, different response times depending on invocations, etc. it is necessary to take data into account from the beginning.

[8] For completeness: if the upper bound ends up over the limit, but the lower bound does not, there is a possibility that things go wrong. If the lower bound goes over the limit, we have the guarantee that the execution is going to violate the initial restrictions. We will assume we do not want to run any risk.

7 Conclusions and Future Work

We have presented a resource analysis for service orchestrations (which we instantiate to the BPEL case) which is based on a translation to an intermediate programming language (Prolog) for which complexity analyzers are available. These analyzers can be customized to deal with user-defined resources, thereby opening the possibility of generating resource-consumption functions, some of them of interest for SOC. Inferring these functions can be used as core technology for some approaches to proactive monitoring, adaptation, and matchmaking.

We sketched the core of the translation process, which approximates the behavior of the original process network in such a way that the analysis results (the cost functions) are valid for the original network. We have sketched a mechanism to use these functions, together with environmental characteristics, to predict the future behavior of the system even when the environment deviates from its expected behavior.

Our translation is partial in the sense that some issues, like correlation sets, are not yet taken into account. A richer translation which we expect will take into account of this (and other) issues is the subject of current work.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Service-Oriented Computing. Communications of the ACM **46**(10), pp. 24–28 (2003)
2. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, BEA, Intalio, Individual, Adobe Systems, Systinet, Active Endpoints, JBoss, Sterling Commerce, SAP, Deloitte, TIBCO Software, webMethods, Oracle (2007)
3. Zaha, J.M., Barros, A.P., Dumas, M., ter Hofstede, A.H.M.: Let's Dance: A Language for Service Behavior Modeling. In: OTM Conferences (1). pp. 145–162. (2006)
4. van der Aalst, W., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In Leymann, F., Reisig, W., Thatte, S.R., van der Aalst, W., eds.: The Role of Business Processes in Service Oriented Architectures. Number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
5. Mukherjee, D., Jalote, P., Nanda, M.G.: Determining QoS of WS-BPEL Compositions. In: ICSOC. pp. 378–393. (2008)
6. Wu, J., Yang, F.: A Model-Driven Approach for QoS Prediction of BPEL Processes. In: ICSOC Workshops. pp. 131–140. (2006)
7. Buccafurri, F., Meo, P.D., Fugini, M.G., Furnari, R., Goy, A., Lax, G., Lops, P., Modafferi, S., Pernici, B., Redavid, D., Semeraro, G., Ursino, D.: Analysis of QoS in Cooperative Services for Real Time Applications. Data Knowledge Engineering **67**(3), pp. 463–484 (2008)
8. Fugini, M.G., Pernici, B., Ramoni, F.: Quality Analysis of Composed Services through Fault Injection. In: Business Process Management Workshops. pp. 245–256. (2007)
9. Cardoso, J.: About the Data-Flow Complexity of Web Processes. In: 6th International Workshop on Business Process Modeling, Development, and Support: Business Processes and Support Systems: Design for Flexibility. pp. 67–74. (2005)

10. Cardoso, J.: Complexity analysis of BPEL web processes. Software Process: Improvement and Practice **12**(1), pp. 35–49 (2007)
11. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems **15**(5), pp. 826–875 (November 1993)
12. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.W.: Lower Bound Cost Estimation for Logic Programs. In: 1997 International Logic Programming Symposium, pp. 291–305. MIT Press, Cambridge, MA (October 1997)
13. Navas, J., Méndez-Lojo, M., Hermenegildo, M.: User-Definable Resource Usage Bounds Analysis for Java Bytecode. In: Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09). Electronic Notes in Theoretical Computer Science. Elsevier - North Holland (March 2009)
14. Canfora, G., Penta, M.D., Esposito, R., Villani, M.: An Approach for QoS-Aware Service Composition Based on Genetic Algorithms. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, New York, NY, USA, pp. 1069–1075. ACM (2005)
15. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. Software Engineering, IEEE Transactions on **30**(5), pp. 311–327 (May 2004)
16. ping Chen, Y., zhi Li, Z., xue Jin, Q., Wang, C.: Study on QoS Driven Web Services Composition. In: Frontiers of WWW Research and Development - APWeb 2006. Volume 3841 of Lecture Notes on Computer Science., pp. 702–707. Springer Verlag (2006)
17. Ivanović, D., Carro, M., Hermenegildo, M., López, P., Mera, E.: Towards Data-Aware Cost-Driven Adaptation for Service Orchestrations. Technical Report CLIP5/2009.0, Technical University of Madrid (UPM) (November 2009)
18. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-Definable Resource Bounds Analysis for Logic Programs. In: International Conference on Logic Programming (ICLP). Volume 4670 of LNCS., pp. 348–363. Springer-Verlag (September 2007)
19. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007). Number 4915 in LNCS, pp. 154–168. Springer-Verlag (August 2007)
20. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Morales, J., Puebla, G.: An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Pierpaolo Degano, Rocco De Nicola, J.M., ed.: Festschrift for Ugo Montanari. Number 5065 in LNCS. Springer-Verlag (June 2008) pp. 209–237
21. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming **58**(1–2), pp. 115–140 (October 2005)
22. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guízar, A., Kartha, N., Liu, C.K., Khalaf, R., König, D., Marin, M., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0. Technical report, IBM, Microsoft, BEA, Intalio, Individual, Adobe Systems, Systinet, Active Endpoints, JBoss, Sterling Commerce, SAP, Deloitte, TIBCO Software, webMethods, Oracle (2007)

Configuring End-to-End Business Processes using Multi-Dimensional QoS Criteria

Qianhui Liang[1], Michael Parkin[2]
Mike Papazoglou[2] and Willem-Jan van den Heuvel[2]

[1] School of Information Systems, Singapore Management University
Singapore 178902, Singapore.
althealiang@smu.edu.sg
[2] European Research Institute in Service Science,
Tilburg University. Tilburg, Netherlands.
{w.j.a.m.vdnHeuvel, m.p.papazoglou, m.s.parkin}@uvt.nl

Abstract. An important area of services research gathering momentum is the ability of a service end-user (or client application) to take a generic business process and configure it according to desired quality of service (QoS) requirements. These QoS requirements may be manifold and be across different logical layers of the application, from business-related to system infrastructure; i.e., they are *multidimensional*. Configuring the generic process with the 'best' services to meet the multidimensional end-to-end QoS requirements of the user from the many services that may be a functional match is a challenging task.

In this paper we propose a pull- or demand-driven approach where functionally equivalent services that fit the generic business process are differentiated on the basis of a variety of "best" or desirable QoS options as the situation demands. The proposed approach considers the optimal choice of multi-dimensional QoS variables using on heuristic algorithms that combine simulated annealing and genetic algorithms and performs an experimental comparison of these algorithms.

Keywords: Service Selection, Differentiated Services, Multi-dimensional QoS, End-to-end QoS, Service Level Agreements.

1 Introduction

An important area of services research gathering momentum is the ability of a service end-user (or client application) to take a generic business process and configure it according to desired non-functional or quality of service (QoS) requirements. Generic business processes like this – also referred to as process skeletons, frameworks, fragments or templates – encapsulate generic know-how about the structural and operational semantics of a particular business process. Their generality refers to their capacity to be changed, tailored or parameterized to enterprise-specific requirements and constraints. By using generic business processes like this we can speed up the creation of composite processes, enable the evolution of enterprise-specific solutions and guide the development of specific service applications by capturing the minimum set of unified concepts, axioms and relationships within the domain-specific problem. Well-known and widely used generic process models include those from RosettaNet, eTOM and SCOR, and the representation of these processes is usually in BPMN, BPEL or other similar notation.

Since the anatomy of a generic business process captures the 'best practice' for achieving a desired business function, its basic structure and behavior are hard-coded (i.e., requirements for the functional composition and integration of elements of that process have already been solved). The tailoring of the generic process is usually performed through configuring the business process with services differentiated by the qualities of service (QoS) they offer. We call services that provide the same functional interface but who offer different levels of service (due to different non-functional characteristics) *differentiated services*.

Using differentiated services when configuring a generic business process into a service-based application, or re-configuring an existing service-based application to meet an agreed SLA, allows the end-user to create/configure the service-based application according to meet a desired *cumulative QoS* for the end-to-end business process. Cumulative QoS signifies a succession of services in possibly different enterprises is either required to be or is successfully integrated into a service-based application that meets the required end-to-end, total QoS for the entire process.

The QoS parameters of a differentiated service, either requested, advertised or agreed (possibly in an Service Level Agreement, or SLA), may be for any logical level of the service-based application, from the (highest) business-level application layer to the (lowest) system-level infrastructure. This is depicted in Figure 1, which illustrates in BPMN a generic end-to-end business process made up of logistics, payment and billing processes (shown as rounded rectangles with dash-dotted borders). These processes contain five component services (rounded rectangles with solid borders) with QoS at two levels of the application: business-related QoS properties, such as on time delivery deadlines and number of active orders, and system-related QoS properties, like process cycle times and process throughput. When a request, advert or agreement for a service contains multi-level QoS like this, we refer to the QoS as *multidimensional QoS*.

A natural approach to differentiating services on the basis of a variety of "best" available or desirable QoS options, either at selection time or when aggregated services are (re)-configured due to a change in requirements, would be a *pull-* or *demand-driven* environment. In this environment end-to-end services could be configured or re-configured according to QoS parameters, service preferences and requirements declared either by the end-user or contained in an agreed SLA. This introduces a novel approach to demand-driven service selection for service-based applications where they are tuned to satisfy end-user or SLA requirements while correlating possibly multiple levels of QoS parameters.

In this paper we propose an approach to treat the on-demand cumulative QoS service selection and configuration problem as an optimization of a utility function that takes into account the relevant constraints specified by the user. The proposed approach considers optimal choice of multi-dimensional QoS variables using the heuristics of simulated annealing, genetic algorithms and a combination of the two.

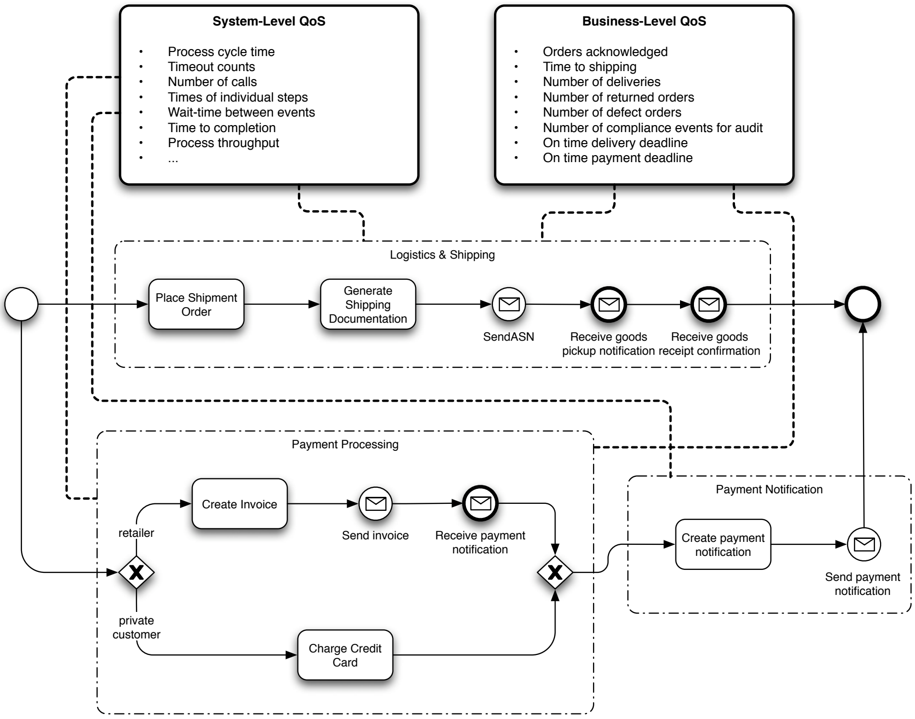


Figure 1: Generic Business Process Example showing Multi-level QoS

In particular, the approach proposed in this paper formalizes a method for optimizing the fitness of an end-to-end service aggregation by representing it as a function over multiple quality dimensions which could be specified in an SLA correlated with the provided QoS of all its component services. We also present a formalization of the constraints associated with QoS parameters and distinguish between hard and soft constraints.

Structure of this Paper: our framework for how the selection/configuration process works is described in Section 2; Section 3 comments on related work in this area; the problem of service selection using differentiated services is formalized in Section 4; Section 5 presents the algorithms we used to evaluate our approach to service selection; Section 6 explains a comparative performance and accuracy evaluation of the algorithms; and, finally, Section 7 presents our conclusions and anticipated future work in this area.

2 Operational Framework

This section explains the process of an end user configuring or re-configuring a service-based application from their formulation of a *request*, and the role of the request, algorithms and utility values they produce in this process.

Figure 2 shows how a request is processed in pseudo-flowchart notation. Note that the generic business process and service metadata repository are assumed to exist before the process starts. If we start with the end-user, which may be an application reconfiguring an existing service-based application because it is failing to meet the terms of an agreed SLA, they formulate a declarative request based on a pre-existing specification of a generic end-to-end business process, as described earlier.

The formulation of such a request involves annotating the generic business process with the end-users desired end-to-end multidimensional QoS requirements. This request can be seen as a constraint on all possible combinations of services that are compatible with the fixed functional composition and integration aspects of the generic business process.

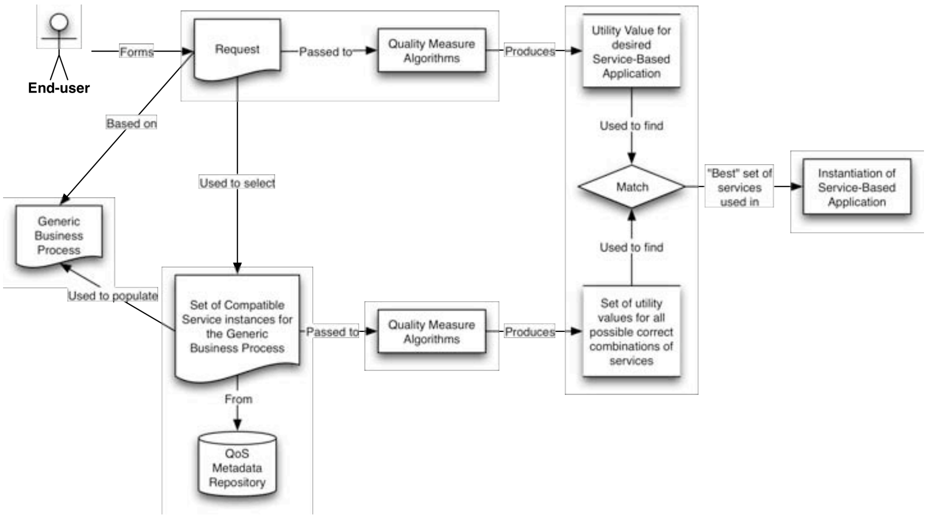
The end-user can create their QoS request for the configured application using 'soft' preferences or 'hard' constraints the application must conform to. An example of a simple QoS preference for the configured system could be that the application processes the order in the minimum amount of time. Simple examples of individual constraints for the system could be that when a application receives an order it should notify the customer before t_1 time units have elapsed that the order has been processed successfully (or not) or that an invoice is sent within t_2 time units. Multidimensional QoS requests contain multiple constraints and/or preferences, such as the generic business process is configured to process at least 100 messages/sec (a system level constraint), process orders at a rate of least 250 orders/hour with maximally 10 defective others (both business level constraints) and/or the application should be configured with services that provide the highest amount of bandwidth possible with the lowest associated processing time and overall cost (system and business level preferences).

The completed multidimensional QoS request is passed to the quality measure algorithm or algorithms to compute the *utility value* of the request (as shown in the upper path in Figure 2). We use the concept of utility value, or the "measure of the total perceived value resulting from an outcome"[1], as it allows us to reduce multidimensional QoS criteria to a single comparable value. Note the utility value is opaque to the end-user; it has no meaning other than it can be used as a means for comparing service configurations.

In parallel with the calculation of the desired utility value, the same request is used to produce a set of utility values for all possible functionally correct permutations of services using service QoS metadata, which could be stored in a UDDI repository or similar service. (This is shown in the lower path in Figure 2.) The resulting set of utility values is then processed and calibrated to determine the optimal matching set of cumulative QoS to the utility value of the request (note that there may not be an

[1] Oxford University Dictionary of Mathematics, Clapham & Nicholson, 2006

exact match, in this case the closest match will be used). The outcome of our approach thus entails a set of QoS levels for the differentiated services in the end-to-end process. In this way an existing service-based application may be (re-)configured with the end-user's desired (or closest possible to the desired) cumulative QoS properties.



End-user
Forms
Request
Passed to
Quality Measure Algorithms
Produces
Utility Value for desired Service-Based Application
Based on
Used to find
Used to select
Match
"Best" set of services used in
Instantiation of Service-Based Application
Generic Business Process
Used to find
Used to populate
Set of Compatible Service instances for the Generic Business Process
Passed to
Quality Measure Algorithms
Produces
Set of utility values for all possible correct combinations of services
From
QoS Metadata Repository

Figure 2: Operational Framework for Configuring a Service-Based Application from an End-User Request

3 Related Work

Recently, research into the non-functional properties of (Web) services has been a focus for the service science community. This has led to the creation of standards, such as WS-Agreement [1], which allows the expression of a service's non-functional QoS, and an SLA to be agreed on the basis of these advertised details.

However, WS-Agreement does not solve the problem of finding a correct match for the user's requirements and a significant amount of work has taken place, mainly in the semantic community, to develop ontologies and languages as models for expressing and/or selecting a service's QoS. An example of this approach is described in [2].

Research effort has also taken place into considering a user's satisfaction level with a service and related business objectives in the form of a global utility function used in the process of service discovery or composition. In both [2] and [3] possible representations of the quality measure for a service aggregation (or request for an aggregation) using Multi-attribute Utility Theory (MAUT) are studied. However, neither [2] nor [3] address possible heuristics for improving the performance of service selection with a proposed representation or model such as a generic business process.

The algorithms we chose to optimize the problem of on-demand cumulative QoS service selection and configuration problem (described in Section 5) have found applications in many other domains. For example, simulated annealing has been used in scenarios from test-data generation [4] to software clustering [5]. Genetic algorithms have also had widespread use in areas from economics to software engineering [6].

Since the generic business process/service-based application approach allows systems to be built by integrating reusable components in an automatic manner, a significant factor here is the speed of service selection based on multiple QoS attributes. Little research has been carried out into this area when considering a large search space (i.e., where there are many functionally compatible services for each step in the generic business process) and [7], [8] and [9] only search for individual QoS attributes where our work searches across many multidimensional QoS attributes.

Generality and variability in software systems has attracted much attention in the domain of Domain Models and Product Line Architectures (PLAs) [10]. Product line architectures constitute a special type of generic business process models that define a group of common products that share a similar set of characteristics within a specific market, e.g., the telecom or the transportation market [11]. In particular [12] describes an algebraic approach to derive products from product lines, similarly to how we would like to form service-based applications from service instances. However, work in that area concentrates *solely* on product assembly using the functional characteristics of product lines. Our work has a different focus as it concentrates on multidimensional non-functional aspects, rather than the single dimension of functional compatibility.

4 Problem Formalization

As described in the introduction, the objective of this research is to satisfy the preferences and constraints of the users when configuring (or re-configuring) a service-based application. In this section, we look at our formalization of the problem of service selection for such systems. The formalization is centered on the non-functional aspects of the system and we model both (soft) preferences and (hard) constraints to maximize the usefulness of the system to its users.

The optimum solution of the problem – i.e., the "best" match to the preferences and constraints of the end-user – is evaluated through a utility function, following the approach described earlier. The value of our utility function is calculated by summing the weighted dimensional utilities contributed by each dimension of each service, which are calculated using the quality measures of the services along a particular dimension.

First, let Q, D, S and T be non-empty sets of normalized quality levels in any service quality dimension, quality dimensions being considered, services available and tasks for the service-based application. Let (1) denote the range of the function on services, where $s[D]$ is the set of quality measures s of the quality dimensions D, and let (2) signify a function that assigns a dimensional quality measure $q_{s,d} \in Q$ to each quality dimension $d \in D$ for each service s ∈ S and all its calls. If p is a partition of tasks such that $p \subseteq T$, then the overall utility, V_b, of the proposed service-based application using binding[2] *b* is given by (3), where *b(p)* is given in (4).

$$s[D] = \{q \in Q \mid \exists d \in D . g(s,d) = q\} \tag{1}$$

$$q : D \times S \rightarrow Q \tag{2}$$

$$V_b = \sum_{d \in D} \left(f_d (\underset{f\ ort \in T, s = b(p).t \in p}{q_{d,s}, \ldots}) * w_d \right) \tag{3}$$

$$b(p) = \{s \in S \mid \exists p . < p, s > \in b\} \tag{4}$$

From (3), we can see the overall utility is determined by three factors; the dimensional quality measures of services and their calls ($q_{s,d}$), how they affect the dimensional utility of that dimension (f_d), and their weightings (w_d). We now discuss $q_{s,d}$ and f_d in more detail.

4.1 Quality Measures

$q_{d,s}$, the quality measure of services and their calls, is calculated using $c_{d,s}$, a one-time, fixed initialization or registration cost associated to the called service, and $k_{t,d,s}$, a floating cost associated with the service call made by a generic business process component service to another component service. We can identify two cases where $q_{d,s}$ takes different forms:

Totally Ordered Quality Dimensions: for totally order quality dimensions, like time and money, $q_{d,s}$ is obtained by summing the costs of all calls specified in the binding plus the one-time cost associated to the called service. $k_{t,d,s}$ is calculated as a sum of the associated service calls made by every task that uses the corresponding service ($t \in T$, $s \in b(p).t$, $p \in P$). In this case, $q_{d,s}$ takes the form shown in (5), with α allowing the same quality dimension to be normalized across all different services.

QoS-like Dimensions: we model these by summing the qualities of each selected task-service match, as shown in (6). Here $\overline{g} : Q \times Q \rightarrow Q$ computes the quality measure by taking into account two measures: the first is the value of $k_{t,d,s}$ associated with the service call made by one of the tasks that use the corresponding service ($t \in T$, $s \in b(p).t$, $p \in P$); the second is the measure $c_{d,s}$ associated with the service. As is usual, we assume that $\overline{g}$ is a polynomial-complexity function.

$$q(d,s) = \alpha \left(c_{d,s} + \sum_{t \in T, s = b(p).t \in p} k_{t,d,s} \right) \tag{5}$$

[2] A binding is defined as an assignment of one or more services to a task partition, according to

$$q(d,s)=\overline{g}\left(c_{d,s},\quad \underset{t\in T,s=b(p).t\in p}{k_{t,d,s}}\right) \tag{6}$$

To give an example of how these quality measures work within the operational framework, if we consider the network bandwidth quality dimension of a service, the bandwidth is constrained by the total bandwidth of the server's network interface and the bandwidth of network at the client side. In this case, $q_{s,d}$ is the minimum of the bandwidth quality of the server and the end-user invoking the service.

5 Algorithms

In this section we present three algorithms to select service implementations that best satisfy an end-users non-functional requirements. In these algorithms we assume the following:

- A representation exists for the service-based application to be built, including a description of the tasks the application must perform and their structural relationships;
- The necessary services to create the service-based application exist for the generic business process and are available;
- We know the QoS preferences and constraints of the end user to be satisfied.

With respect to their operation, each of the algorithms returns the collection of services as a binding that can be used to form a system of a heuristically good (large) enough utility to its users.

5.1 Simulated Annealing

Simulated annealing (SA) is the name for a class of algorithms that can be used to find the global minimum or maximum of a function in a large, multidimensional search space. We use an SA algorithm to perform a search in the set of bindings in order to find a binding with the largest utility.

For reasons of space we do not describe the full details of the simulated annealing approach here, for more information on the SA algorithm the interested reader is directed to [13]. Much of our SA algorithm remains unchanged from that in [13], however some problem-specific considerations we have taken into account include the representation of the solution space, how local moves (from one binding to another where the assignment to one task in the original binding is switched) are performed, and the formalization of "best" solution (these are described in Section 4). As an optimization we also preprocess the service metadata to remove services instances from the search that have low average quality values or violate any of the

the service constraints. An alternative definition is that a binding is a "best" match between a end-user requested QoS for a set of tasks and a QoS advertised by a service.

constraints. The remaining services are passed to the SA algorithm. When the algorithm completes, it outputs the best utility value and its associated binding (the configured service-based application configuration) for the given request.

5.2 Genetic Algorithm

The second algorithm we present is a Genetic Algorithm (GA). As with the simulated annealing algorithm, much of our implementation is standard and we do not present it in detail; the interested reader is directed to [14] for a full treatment. However, within this genetic algorithm our contribution is to use a context free grammar (CFG) to represent the set of tasks and control structures in the service composition to encode the genome.

We designed a CFG so we can represent the tasks and control tasks as a string, similarly to how a genetic sequence is encoded. If t_1, t_2, …, t_i, t_j denote tasks in the task set, we can represent the control structures in BPEL as terminal characters:

- *Q* : <sequence>
- *I* : <if>
- *L* : <elseif>
- *E* : <else>
- *R* : <repeatUntil>
- *W* : <while>
- *F* : <flow>
- *P* : <pick>

We also define *A* and O to represent logical “and” and “or” operators, and *C*, *r* and *s* to represent constant values, relational operators and the starting value of the algorithm. Assuming the rules of the grammar are as shown in (7) and *Q, I, L, E, R, W, F, P, A, O, C, r*, t_n, <, >, ≤, ≥ are terminal symbols and *e*, c_n, *l, r, o* and *s* are non-terminal symbols.

The third rule from the bottom tells that the condition of enacting a task may be recursively determined by logical combinations of a task satisfying certain Boolean relations with a constant. This rule affects the conditional structure of <if>, <elseif>, <else>, <repeatUtil>, <while> and <pick>, as shown in the second and fourth rule. The second rule tells that a system can be configured for a) a single task, b) recursively connecting tasks connected by sequential and parallel structures or c) connecting a conditional expression and a task.

$$\begin{aligned}
&s \to e\\
&e \to t_i \mid c_1 e t_j \mid c_1 t_i e \mid c_2 l e\\
&c_1 \to Q \mid F\\
&c_2 \to I \mid L \mid E \mid R \mid W \mid P\\
&l \to r t_i C \mid o l^2\\
&r \to \le \mid \ge \mid < \mid >\\
&o \to A \mid O
\end{aligned} \tag{7}$$

The fitness of the generation is defined to be the utility function value of the individual service minus a penalty for any unsuitable QoS factors. It is shown in (8).

$$F_b = V_b - w_g * \frac{g}{g_{\max}} * \sum_i \frac{dc_{d,\Omega,i}}{R_{\max} - R_{\min}} \tag{8}$$

$dc_{d,\Omega,i}$ the amount that the individual exceeds the upper or lower bound specified in i^{th} constraint, where i is the index to the constraint. R_{max} and R_{min} are the upper and lower bounds of the constraint. g is the index to the current genome generation and g_{max} is the maximum number of generations allowed. w_g can be used to adjust the weighting of the penalty. The penalty is determined by two factors: where the current generation is in the evolution process and the amount of the 'negative' impact on the service's utility due to its unsuitability. Early in the genome's evolution, g/g_{max} is small and a smaller penalty will apply due to the native impact of the unsuitable subpopulation. Later in the evolution process, g/g_{max} increases and a greater penalty will be given due to the unsuitability. If the unsuitability of a service is only a small amount over the threshold or the unsuitability is caused by a violation of one or a small number of constraints, the penalty is relatively small.

5.3 Hybrid Algorithm

The hybrid algorithm appends a simulated annealing algorithm to the end of the genetic algorithm. In this scenario, the simulated annealing algorithm attempts to locate (local) optima based on the result from the final population in the genetic algorithm. The purpose of such hybrid is to get the best of both worlds – i.e., the accuracy of the genetic algorithm with the performance of the simulated annealing.

In the algorithm, the variables and their significance are explained in Table 1 and the pseudocode for the hybrid algorithm presented in Figure 3.

Table 1: Variables used in the hybrid algorithm

Variable	Explanation
Fitness	A function that assigns a fitness score to a binding.
n	The number of individuals in a population
R_C	The threshold rate of the population individuals to apply crossover operation
R_M	The number of individuals to apply mutation operation
Max_G	The maximum number of generations allowed
Max_I	The maximum number of iterations allowed for hill climbing in the last population
Max_R	The number of reselections allowed for selecting a task partition for local move

6 Experimental Verification

This section describes the performance results of a comparison of implementations of the three algorithms from Section 5. All experiments were performed on a Windows XP PC with 512MB of RAM and a 2.86GHz Pentium P4 processor. The algorithms used in the experiments were written using the Sun's Java 6.

Before describing the experiment and its results the notation used in the analysis is shown in Table 2. Most of the symbols have an intuitive meaning, but PH needs more explanation since it is the number of possible execution paths in the generic business process due to conditional branching. As a simplifying assumption for our initial experiments, when a conditional branch appears in a generic business process we assume an equal probability in taking any branch.

```
for each part $p$ in task partition P do {
    With the most recent knowledge, remove the assignments that cannot be part of the result to be output
    Generate $n$ bindings randomly and put them in $G$;
}
$F^*_b = 0$
for each binding $b$ in $G$ do {
    Calculate $F_b$
    If $F_b > F^*$, $F^* = F_b$;
}
for $i$=1 to Max_G do {
    Probabilistically select (1 - R_C) bindings in $G$ with a probability of $F_i/\sum_{k=1,n} F_k$ for the $j^{th}$ binding and add them to $G_{new}$
    Probabilistically select (R_C * n/2) pairs of individuals from $G$ and produce two children from each pair by applying the crossover operator
    Add all offspring to $G_{new}$;
    Randomly select (R_M * $n$) bindings in $G_{new}$ and apply the mutation operator;
    $G = G_{new}$;
    for each $b$ in $G$ do {
        L1:
        Calculate $V_b$
        $V^{**} = (V^* = V_b)$
    }
}
for $i$=1 to Max_I do {
    $j$=0;
    L2:
    Select one part $p$ in $P$ randomly
    Randomly select another service $S'$ for $p$ such that $<S', p> \in M$
    Swap service assignment S of tasks in $p$ within $b$ with $S'$ and form $b'$
    if $b'$ is feasible {
        Compute $V_{b'}$
        if $V_{b'} \leq V^*$ {
            if $\exp(V_{b'}-V^*)/T_c >$ rand(0,1) {
                goto L3;
            }
            else {
            L3:
            $V^* = V_{b'}$ and $b = b'$
        }
    }
    else {
        $r$ = rand(0,1)
        if ($r$ < Rate) goto L3
    }
    i++
}
if ($tc > T_n$) {
    $tc$ *= coolingRate;
    $i$=1
    goto L2
}
if $V^* > V^{**}$ {
    $V^{**} = V^*$ and $b^{**} = b^*$
}
Remove $b$ from $G$
if $G$ is not empty goto L1
Return binding $b^{**}$ and $V$
```

Figure 3: Pseudocode for the hybrid algorithm

Table 2: Symbols used in analysis of experiment results

Symbols	Meaning
A_V	The average utility value of all the experiment runs for a particular problem instance
N	The number of component services in the end-to-end process
PH	The total number of total execution paths in the end-to-end process
S	The total number of service implementations matching the functional request
T	The average completion time (ms) for the algorithm
V	The maximum utility value of all the experiment runs for a particular problem instance

Table 3: Initial settings for each algorithm

Parameter	Simulated Annealing (SA)	Genetic Algorithm (GA)
Cooling rate	0.99	0.9
T_0	100	10
T_n	1000	100
Rate	0.3	0.03
Max-iteration	10	10

Table 4: QoS constraints considered

Dimensions	A	B	C
Constraints	Time	Time+Cost	Bandwith+Time+Cost

To investigate the performance and accuracy of the simulated annealing and genetic algorithms, our first experiment was to find the time to select the "best" services for a requested configuration of the generic business process shown in Figure 1 from a set of functionally compatible services. The end-user request used in the experiment was to configure the generic business process with the highest bandwidth possible with the lowest associated processing time and cost. For the example shown in Figure 1, the starting values for the algorithms were as shown in Table 3. The value of S for this example is 5 (as there are five component services) and PH is 3 (as there are three possible execution paths).

The calculation of the utility value for each functionally correct combination of services is performed along the following dimensions; the minimum time, the minimum time and cost and the minimum time and cost but highest bandwidth. These QoS constraints are shown in Table 4. We investigated two cases where, for each component service of the generic business process, the "best" service is chosen from either five or ten service instances constrained by the QoS in Table 4. Table 5 shows the results. In these results a higher value of utility here means the algorithm is more accurate in finding the minimum/maximum preferences of the user. As can be seen, we found the simulated annealing to be quicker than the genetic algorithm in both cases. However, the genetic algorithm yields a higher (better) utility than simulated annealing in both cases.

Table 5: Performance and accuracy with various numbers of services for 5 tasks

N	PH	S	T		V	
			SA	GA	SA	GA
5	3	5	24	33	8362	8998
5	3	10	1479	3726	5329	7380

To study the performance of the hybrid algorithm with respect to the simulated annealing and genetic algorithms, we performed further comparison experiments in more realistic settings that exhibit more complexity. These experiments used the same QoS constraints as before, but instead of using the generic business process example from Figure 1, we created datasets representing generic business processes as input to the algorithms with each dataset containing different (random) numbers of component services and conditional branches but with a constant number of services instances to select the “best” from.

To compare the performance of the hybrid algorithm we calculated the ratio of the time required to finish the simulated annealing and genetic algorithms and the hybrid of the two. Tables 6 and 7 show the results of our comparison and show the ratio of the corresponding results of two algorithms calculated using the running time, *T*, maximum utility, *V*, and average utility, A_V (calculated over 10 iterations of the algorithms).

In general, we found that the hybrid of the simulated annealing and genetic algorithms improved the utility value (i.e., the accuracy) and the hybrid finds the utility value faster than the genetic algorithm. Of course, we found several abnormal cases in our experiments; for example, with for 8 tasks and 81 functional matches (not shown in these results) the time to complete the hybrid is more than that of the genetic algorithm, but the hybrid’s utility value is much smaller (i.e., less accurate) than the genetic algorithm’s value.

Table 6: Average result comparisons on time, utility

N	PH	S	Ratio T		Ratio V		Ratio A	
			HA/SA	HA/GA	HA/SA	HA/GA	HA/SA	HA/GA
5	384	5	1.21	0.88	1.07	1.00	1.24	0.94
6	90	5	1.09	0.91	1.59	1.03	1.41	1.01
7	108	5	1.09	0.91	0.99	1.11	1.03	1.11
8	960	5	1.11	0.85	1.02	1.02	1.07	1.01
9	270	5	1.44	0.80	1.01	1.00	1.01	0.97
10	786	5	1.39	0.74	1.00	1.00	1.22	1.01

Table 7: Average result comparisons on time, utility

N	PH	S	Ratio T		Ratio V		Ratio A	
			HA/SA	HA/GA	HA/SA	HA/GA	HA/SA	HA/GA
5	39200	10	1.77	0.75	1.42	1.05	1.38	1.02
6	504	10	1.05	0.82	1.54	0.94	1.71	0.91
7	65856	10	1.68	0.88	1.07	0.97	0.95	1.03
8	81	10	1.94	1.02	1.41	0.86	1.02	0.93
9	60	10	5.59	0.78	1.00	1.00	1.10	0.96
10	720	10	76.30	0.47	1.04	1.11	1.51	0.86

To surmise our results, our experiments have shown that, for our experimental configuration and choice of parameters, the genetic algorithm is substantially more accurate than the simulation algorithm, although it takes longer to achieve this accuracy. The simulated annealing algorithm is faster, but has a much lower accuracy than the genetic algorithm. The hybrid algorithm using the genetic and simulated annealing algorithms shows the trade-off in performance and accuracy; it achieves accuracy higher than running the simulated annealing algorithm by itself, yet has only a small improvement in accuracy over that of running the genetic algorithm. The results of the hybrid are understandable because genetic algorithm plays the more significant role in the hybrid and, thus, the final result tends to the results of that genetic algorithm. Regarding execution time, the hybrid is slightly more expensive than both the simulated annealing and genetic algorithms.

7 Conclusions & Future Work

The ability of a service end-user to take a generic business process and configure it according to desired quality of service (QoS) requirements constitutes an important and challenging problem for services research. This paper presents a demand-driven operational framework where users are enabled to configure generic business processes and generate customized or differentiated services that are parameterized in terms of multidimensional QoS information. This is a novel approach that goes well beyond conventional approaches to configuring generic business processes according to purely functional characteristics.

The proposed approach is based on a twin-track procedure where utility value is calculated for any given user request and is compared with a large number of potentially matching business processes that may satisfy the stated QoS requirements. The approach is based on optimization techniques that operate on the fusion of two proven algorithms (a simulated annealing algorithm, a genetic algorithm). The approach has been explored and validated by an initial experimental implementation, which will be fine-tuned and extended in the future and in particular will deal with probabilistic data about execution paths which can be gathered from event logs.

This approach is generic in nature in that it applies equally well to different levels of process granularity, for example, business processes within end-to-end aggregations, component services within individual business processes, or even to activities within component services.

The results in this paper are preliminary in nature and refinements and extensions are needed in several directions. Initially, we wish to further validate our approach with additional experiments to further calibrate the initial parameters of and the weights attached to the simulated annealing and genetic algorithms in the hybrid algorithm. Following this, we intend to develop a prototypical toolset to provide integrated support all constituents of the operational framework we have outlined, and enable the end-user to configure their generic business processes to optimally suit the problem at hand.

References

1. A. Andrieux *et. al*. Web Services Agreement Specification (WS-Agreement). Recommended Standard, Open Grid Forum, March 2007. Grid Resource Allocation Agreement Protocol Working Group (GRAAP-WG).
2. Q. Liang, JY Chung, and S. Miller, “Modeling Semantics in Composite Web Service Requests by Utility Elicitation”, Knowledge and Information Systems (KAIS) Journal, 2007.
3. Y.J. Seo, H.Y. Jeong, Y.J. Song, “A Study on Web Services Selection Method Based on the Negotiation Through Quality Broker: A MAUT-based Approach”, ICESS 2004: 65-73.
4. N. Tracey, J. Clark, and K. Mander, “Automated Program Flaw Finding Using Simulated Annealing”, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98), March 1998.
5. Y. Zhang, M. Harman, and S. A. Mansouri, "The Multi-Objective Next Release Problem", Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), 2007.
6. J. J. Dolado, “A Validation of the Component-Based Method for Software Size Estimation”, IEEE Engineering, 26(10):1006–1021, 2000.
7. P. A. Bonatti, P. Festa, “On Optimal Service Selection”, Proceedings of 14th International Conference on the World Wide Web, 2005.
8. R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz, “Heuristics for QoS-aware Web Service Composition”, In Proceedings of the IEEE Conference on Web Services, 2006.
9. D. Menasce, E. Casalicchio, V. Dubey, “A Heuristic Approach to Optimal Service Selection in Service-Oriented Architectures”, 7th ACM Int. Workshop on Software and Performance (WOSP 2008). June 2008.
10. L. Trat, “The MT model transformation language”, Proceedings of SAC’06, ACM, 2006
11. S. Sendall and W. Kozaczynski, "Model Transformation - the Heart and Soul of Model-driven Software Development", IEEE Software, 20(5) pp. 42-45, Sept/Oct 2003
12. A.W. Scheer, “ARIS- Business Process Frameworks”, Springer, Third Edition, 1999
13. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, “Optimization by Simulated Annealing”, *Science* 220, pp 671-680. 1983.
14. M. Mitchell, “An Introduction to Genetic Algorithms”, The MIT Press. ISBN 0262631857, 1998.

A CMMI Based Configuration Management Framework to Manage the Quality of Service Based Applications

Sajid Ibrahim Hashmi, Stephen Lane, Dimka Karastoyanova[1], and Ita Richardson

Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland
{sajid.hashmi, stephen.lane, ita.richardson }@lero.ie
IAAS, University of Stuttgart, Germany[1]
dimka.karastoyanova@iaas.uni-stuttgart.de[1]

Abstract

Service Based Applications (SBAs) have highlighted new challenges related to Configuration Management (CM) which is an important process for the assurance of end to end quality in software systems [1]. As far as the quality of SBAs is concerned, configuration management remains an issue because of the loosely coupled and adaptive nature of the corresponding applications. A smart configuration management approach will allow organizations to make their IT resources more reliable and to utilize them to their maximum. In this paper, we propose a service-based configuration management framework based on SEI-CMMI-SVC [6] which contributes to the S-Cube project [reference] Implementing this approach will allow organisations to effectively manage the configurations of their SBAs.

Keywords: Service Oriented Architecture, SBAs (Service Based Applications), Configuration Management (CM), Quality Assurance, CMMI (Capability Maturity Model Integration)

1 Introduction

Today’s computer world consists of applications which are scattered across different networks and require special effort in terms of integration. Developers of such applications need to pay special attention to configuration in order to ensure smooth operation. Managing the overall configuration of an IT system is important as the Enterprise Management Associates has noted that 60 percent of service impacts are due to configuration problems [1]. There are many issues related with poor Configuration Management (CM) of software configurations; such as system related failures, failure of key services, deficiency in performance, reduction in employee productivity, all of which consequently can cause serious business impact. These are some recommendations for efficient CM [2]:

1. Control configuration items and the way they change.
2. Use technology to help discover, record, and maintain configuration information.
3. Configuration management implementation must enable change, release, and problem management.

In service-oriented environments the heterogeneity of resources is dealt by virtue of providing any kind of functionality or resource as a service with a stable interface. However this does not completely remove the need for configuration of the resources, which has to be performed by any service provider of an SBA. The configuration of service implementations and resources must therefore comply with the global requirements of the SBA and must be met at the site of each of the distributed pieces of software. Any kind of local configuration management has implications on the service quality and functional properties, which can have undesirable consequences.

1.1 IT Infrastructure

Organizations have business processes in place in order to meet their objectives; e.g. sales, administration, and financial departments work together in a “Sales” process. Each of the units involved in an organization needs one or more services (e.g. application softwares or utilities). These services run on IT infrastructure which includes both hardware and software, therefore it must be managed accordingly to meet organizational objectives [3]. Proper management of IT infrastructure will ensure that the required services by business processes are available. CM is part of this IT infrastructure which consists of procedures, policies, and documentation.

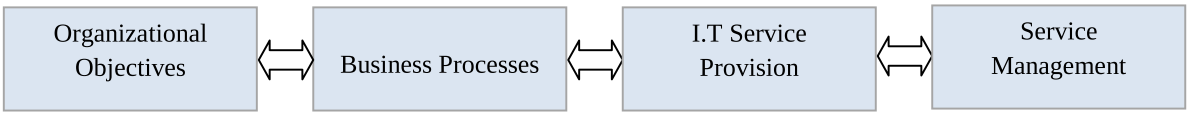


Fig. 1 IT Service Management

1.2 Software Quality Assurance

Software quality assurance is about identifying the right things to implement and test, and allocating and managing resources in a way that minimizes risks when applications and services are deployed [4]. There are two types of quality assurance activities: constructive and analytic quality assurance. As the name suggests the purpose of the first one is to prevent fault injection while artifacts are created whereas the other one deals with cleaning artifacts after they have been developed. In this research we aim to support constructive quality assurance because rework often increases the associated cost and overhead of a software project.

CM is a quality enabling process which provides a logical view of services by identifying, maintaining, and verifying the versions as well as the corresponding configuration items [2]. The objective of this research is to create a CM framework that can contribute to the end-to-end quality assurance of SBAs. While there are many ways to assure the quality of a software system, such as software testing, the use of CASE tools or the implementation of software development best practices and process implementation, the CM process was chosen because having the configurations of SBAs effectively managed would go a long way towards assuring their quality. In terms of end to end quality, a CM process would allow developers more accurately develop and update the correct versions of services. SBAs or other service consumers would also benefit from CM as they would see a new version numbers when services get updated, this would then allow them to react to updates if necessary. A CM process for services or SBAs would need to have some proactive element to inform consumers of new versions to prevent downstream incompatibilities.

2 Background

2.1 S Cube

The objective of the S-Cube project is to create an integrated European research community in the area of software and service engineering. It is based on an ideology that the engineering and management of SBAs is quite different than traditional software applications as they are built by combining different services which may be provided by third parties with whom there should be a service level agreement.

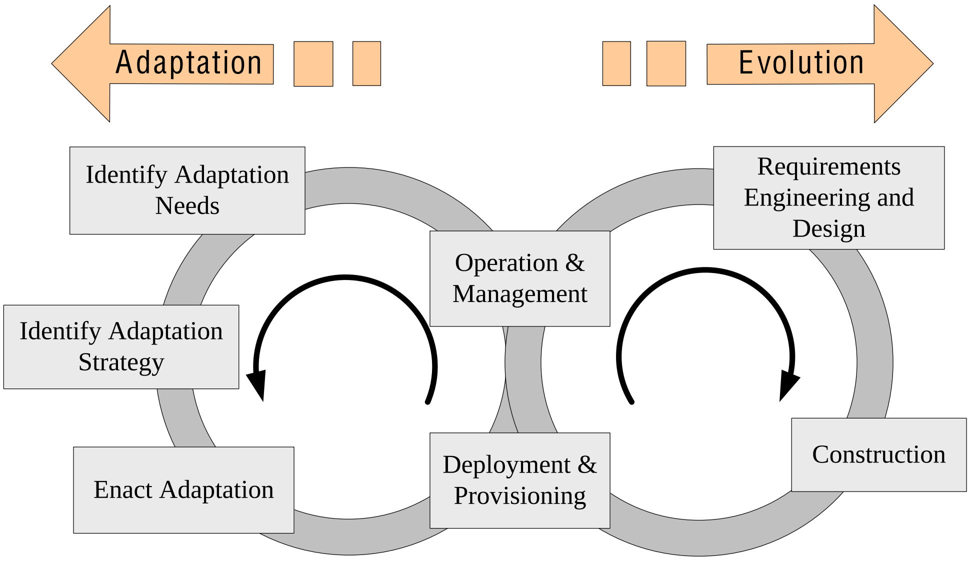

Adaptation
Evolution
Identify Adaptation Needs
Requirements Engineering and Design
Operation & Management
Identify Adaptation Strategy
Construction
Enact Adaptation
Deployment & Provisioning

Fig. 2 S-Cube Reference Life Cycle [10]

Figure 2 describes a reference lifecycle for SBAs which has been proposed by the S-Cube project [10]. It is composed of two main cycles: the evolution cycle depicts classical application design; i.e. requirements engineering, design, coding, and deployment while the adaptation cycle reflects the adaptation of SBAs. SBAs need to accommodate many changes at run time and this two cycle approach allows a good balance between the design and runtime operation. The operation and management phase belongs to both phases. Therefore, it must be efficient and precise enough to meet the transition needs of the entire life cycle. By improving the CM process within operations and Management, we aim to strengthen the entire life cycle of S-Cube.

2.2 Configuration Management

Software Configuration Management (SCM) is a Software Quality Assurance (SQA) process for managing different configurations of configurable software items. (Galin, 2003). Many items change during a software products lifetime and it is important to keep track of these changes for a number of important reasons. One reason for a good configuration management process is to facilitate customer support. Customers may have different software versions so it is important to know which version each customer is using in order to support them effectively. For customers support queries it may be necessary to easily access various version of source code, design documents or support documentation.

In component based development (CBD) software applications can be made up from several standalone components [12]. In CBD, it is important to record the version of each component included in each application release. If the versions of the components are changed then the version of the overall application should subsequently change. It is important for software developers to record configuration changes in order to facilitate the quality assurance of the entire software system. In traditional software systems configuration management can be achieved successfully if a suitable process guideline or standard is followed. Once such standard for this area is IEEE 828:2005, the IEEE standard for software configuration management plans [13]. Alternatively a guide such as Leon's[14] guide to software configuration management can be followed.

2.3 CMMI-SVC

Capability Maturity Model Integration (CMMI) [5] models are a collection of best practices that help organizations to improve their processes. This process improvement can be implemented by means of either Continuous or Staged representations of CMMI. Continuous representation allows organizations to select a specific process area and improve relative to it by means of satisfying it's goals and practices. On the other hand Staged representation allows organization to earn a process maturity level when all the related process areas have been successfully implemented. CMMI - SVC [6] is a CMMI assemblage that covers the activities designed to manage, establish, and deliver services. It has been designed for service industry as a process improvement framework and its goals and practices are relevant to any organization concerned with delivery of service. CMMI – SVC includes 25 process areas split by 4 process categories. We make use of Expert Judgement [11] to utilize only those process areas and subsequent practices which can support CM practices. We have opted for continuous representation of CMMI since our goal is to improve the CM process instead of achieving a maturity level for the organizations; also this representation allows more flexibility when using the model.

3 Research Methodology

We propose a CM framework specifically designed with the adaptation of SBAs in mind. The starting point for this framework is the textbook description of the CM process as outlined in Galin [7]. This process description was designed for traditional software development and has many deficiencies in relation to service-oriented development. Galin's CM process description is comprised of a set of configuration activities and their associated action items. Therefore, in order to solve this problem, practices from the CM and other supporting process areas within CMMI-SVC were used to support the action items from Galin's CM process. It is worth considering whether or not to use the CMMI-SVC CM process area directly rather than using parts of it as an add-on to Galins' CM process. However, since CMMI-SVC is targeted at general services and not specifically at Service-Oriented Computing (SOC) or even software, all of its practices are not directly applicable to SOC. Therefore a suitable methodology was to use the skeleton of a Traditional Software Engineering (TSE) CM process and supplement it with software applicable practices from CMMI-SVC.

Figure 3 shows the configuration management activities identified by Galin [7], which form the starting point for our proposed CM framework. Four levels of CM activities are defined which further contains set of certain action items.

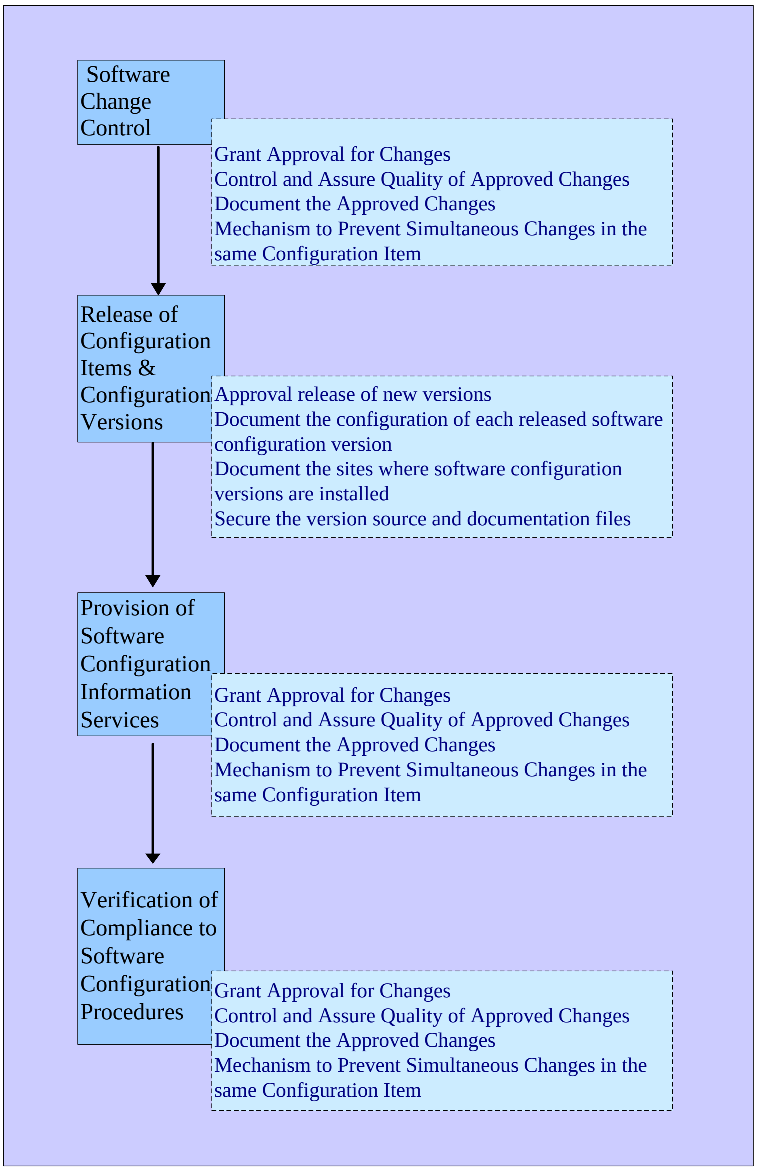

Software Change Control
Grant Approval for Changes
Control and Assure Quality of Approved Changes
Document the Approved Changes
Mechanism to Prevent Simultaneous Changes in the same Configuration Item
Release of Configuration Items & Configuration Versions
Approval release of new versions
Document the configuration of each released software configuration version
Document the sites where software configuration versions are installed
Secure the version source and documentation files
Provision of Software Configuration Information Services
Grant Approval for Changes
Control and Assure Quality of Approved Changes
Document the Approved Changes
Mechanism to Prevent Simultaneous Changes in the same Configuration Item
Verification of Compliance to Software Configuration Procedures
Grant Approval for Changes
Control and Assure Quality of Approved Changes
Document the Approved Changes
Mechanism to Prevent Simultaneous Changes in the same Configuration Item

Fig. 3 Configuration Management Activities [Galin]

3.1 Research Results

The Operations and Management phase is extended by us to accommodate the CM activities in it (refer to Figure No. 4). We aim to strengthen the entire life cycle of S-Cube by improving the CM process within this phase. As part of our research methodology, we identify the CM activities and all the corresponding action items in them. For each action item, we find corresponding CMMI-SVC practices within its process areas which can support the implementation of these actions items. This one by one mapping of CM supporting action items with relevant CMMI - SVC process areas' practices gives birth to a CM frame work for SBAs.

3.2 The Framework

In order to implement the four activities: Software Change Control, Release of Configuration Items (CI) and Software Configuration Versions, Provision of CM Information Services, and Verification of Compliance to SCM Procedures we map them against a set of CMMI process areas and practices in Tables 2 to 5. Some of the process areas we map are from CMMI which is the basis for CMMI-SVC. Since services are also software applications, CMMI practices are also appropriate. In our mapping, a process area or practice may be used multiple times to implement different CM activities. Table 1 illustrates the first level of our framework; it displays the mapping between the action items for Software Change Control and appropriate practices from CMMI-SVC and CMMI.

Action Items (from Galin)	Relevant CMMI – SVC Process Areas	Corresponding CMMI Practices
Grant Approval for Changes	• Service System Transition	• Analyze Service System Transition Needs (SP 1.1) • Prepare Stakeholders for Changes (SP 1.3)
	• Strategic Service Management	• Gather and Analyze Relevant Data (SP 1.1)
	• Configuration Management • Requirements Management	• Establish change management system (SP1.2) • Manage Requirements Changes (SP 1.3) • Maintain Bidirectional Requirement Traceability (SP 1.4) • Identify Inconsistencies between Project and Requirements (SP 1.5)
Control and Assure Quality of Approved Changes	• Process and Product Quality Assurance	• Objectively Evaluate Processes (SP 1.1) • Objectively Evaluate Work Products (SP 1.2) • Resolution of Non-compliance Issues (SP 2.1) • Establish Records (SP 2.2)
Document the Approved Changes	• Configuration Management	• Establish CM Records (SP 3.1) • Perform Configuration Audits (SP 3.2)
Mechanism to Prevent simultaneous changes in the same SCI by more than one team	• Configuration Management	• Track Change Requests (SP 2.1) • Control Configuration Items (SP 2.2)

Table 1. Software Change Control

The first CM activity in the framework we have put forward is software change control. This is an important activity which ensures changes to software systems are carried out with appropriate levels of governance. This prevents unappropriated or unsafe changes form being made without approval; and this becomes particularly important in SOC where changes to services may affect many downstream SBAs. In order to implement the action items of this activity suitable practices were taken from the Configuration Management, Requirements Engineering, and Process and Product Quality Assurance process areas of CMMI-SVC and CMMI. The Configuration Management and Requirements Engineering processes areas provided practices for the steps required to implement software change control, while the Process and Product Quality Assurance process area provided practices for quality assurance during this process. The following are descriptions of the practices within these process areas can support the CM process.

1. *Analyze Service System Transition Needs* is a practice used to to analyse the functionality of current and the future service system, so that transition related issues can be identified and addressed.
2. *Prepare Stakeholders for Changes* prepares all the stakeholders to embrace change to avoid impairment of service system transition.
3. *Establishing Change Management System* involves creating a change management system that includes storage media, procedures, and tools for recording and accessing change requests. Grant for approval should only be allowed when there is a procedure already laid down to accommodate the change.
4. *Manage Requirement Changes* analyzes the impact of changes, it is important to manage these additions and modifications accordingly.
5. *Maintain Bidirectional Requirement Traceability* ensures that all desired changes have been completely addressed. Requirement traceability also helps to maintain relationship between work products, design, and test plans.
6. *Identify Inconsistencies between Project and Requirements* helps to identify any inconsistency in case part or whole of the requirements are in collision with project plan and/or work products.
7. *Objectively Evaluate Processes* helps finding out the gaps between actual and performed software processes; this information helps to identify if there are going to be any gaps associated with development of new versions.
8. *Resolution of Non-compliance Issues:* is used while evaluating existing configuration items, those features or requirements in the upcoming release can be identified which are not adherent to process, description, procedures, or applicable standards.
9. *Establish Records* requires the maintenance of records for quality assurance reports, corrective actions, and evaluation logs certainly helps during forthcoming releases to take benefit from.
10. *Establish CM Records* helps to remain integrity because separate record keeping of each configuration item helps to recover previous versions which assist in identifying any kind of anomaly between current and potential application baselines.
11. *Perform Configuration Audits* confirms that the resulting baseline would conform to existing standards.
12. *Track Change Requests* can be helpful because you can identify the corresponding effect a change or a new requirement may have. It can also help you to identify a feasible set of requirements in the forth coming baseline.
13. *Control Configuration Items* ensures that only those configuration items are in the baseline which is necessarily approved.

The second activity in the framework as illustrated in Table 3 is the release of software configuration items and software configuration versions. When new software versions are released it is important to record version details and where versions are installed. This information is needed to assist with trouble shooting and diagnosing software errors. With regard to services, the recording of installation sites is not usually an issue as they are usually installed in one location with multiple applications accessing the same services. The release of software configuration items and software versions have different implications depending on whether services or SBAs are being considered. When new versions of services are released it is important to have access to details of previous versions in the event that they need to be reverted to a previous version. This may be necessary of there is an incompatibility issue with a service consumer. As an alternative to reverting back to a previous version it may be necessary to publish multiple versions of the same service concurrently to satisfy all service consumers. When SBAs are considered a new application version may be released by adding services or removing services from an existing SBA. Similarly an SBA may get a new version if its component services are updated to a new version number. In either of these cases it is important to record configuration details and document version releases after releases have been approved. The documentation and source code for each release is and important resource for support and quality assurance activities. This activity can be achieved in SOC using practices form CMMI-SVC activities such as Configuration Management, Project Monitoring & Control, and Process and Product Quality Assurance.

Action Items (from Galin)	Relevant CMMI – SVC Process Area (s)	Corresponding CMMI Practice(s)
Approval release of new versions	• Configuration Management • Project Monitoring & Control	• Control Configuration Items (SP 2.2) • Monitor Commitments (SP 1.2) • Conduct Progress Reviews (SP 1.6) • Conduct Milestone Reviews (SP 1.7) • Analyze Issues (SP 2.1) • Take Corrective Actions (SP 2.2) • Manage Corrective Actions (SP 2.3)
Document the configuration of each released SC version	• Process and Product Quality Assurance • Configuration Management	• Objectively Evaluate Work Products (SP 1.2) • Communicate and Ensure the Resolution of Noncompliance Issues (SP 2.1) • Identify Configuration Items (SP 1.1) • Establish change management system (SP 1.2) • Establish Configuration Management Records (SP 3.1)
Document the sites where SC versions are installed	• Configuration Management • Service Delivery	• Establish change management system (SP 1.2) • Prepare for Service System Operations (SP 2.2)
	• Service Delivery	• Receive and Process Service Requests (SP 3.1)
Secure the version source and documentation files	• Configuration Management	• Establish Configuration Management Records (SP 3.1)

Table 2. Release of SCI and Software Configuration Versions

The following are descriptions of the CMMI practices used to implement the Release of SCI and Software Configuration Versions activity.

1. *Monitor Commitments* identifies the commitments which are satisfied and document the results, and the reason if they are not satisfied.
2. *Conduct Progress Reviews* performs project reviews at certain times to keep stakeholders informed about the project status, so that issues and performance short falls within the baseline can be identified and updated.
3. *Conduct Milestone Reviews* performs milestone reviews to investigate project's accomplishments. It will surely help identifying and updating baseline for problematic features or requirements.
4. *Analyze Issues* identifies and solves issues where there is a deviation from a configuration baseline.
5. *Manage Corrective Actions* manages corrective actions until their completion to obtain the desired results.
6. *Objectively Evaluate Work Products* performs evaluation of work products against standards and procedures. A clear evaluation criterion would help to maintain a perfect baseline.
7. *Identify Configuration Items* identifies configuration items and work products which are designated for configuration management.

8. *Prepare for Service System Operations* makes sure that all configuration items and baseline is ready for services to operate. Corrective actions must be performed repeatedly to ensure consistent service delivery.
9. *Receive and Process Service Requests* involves services requests being made through web and by person.

The next activity in the framework is the provision of software configuration management information. Action items for this activity can be seen in Table 4. This activity supports the first two activities which are about the documentation and control of configuration management. After taking the time to implement a CM process and record data it is important to present this data as useful information for all the stakeholders in the software project. It is important that this information is disseminated at appropriate times such as when milestone versions are released or when major configuration changes take place. Practices for this activity have been taken from the Project Planning and Project Management and Control process areas from CMMI-SVC.

Action Items (from Galin)	Relevant CMMI – SVC Process Area(s)	Corresponding CMMI Practice(s)
Information about the status of changes	Configuration Management Service System Development	Track Change Requests (SP 2.1)
Information about versions installed at a site as well as about the site itself	Service System Transition	Deploy Service System Components (SP 2.1)
	Service System Development	Ensure Interface Compatibility (SP 2.3) Validate the Service System (SP 3.4)
Version history list	Configuration Management	Control Configuration Items (SP 2.2)
Accurate copies of given versions	Configuration Management	Control Configuration Items (SP 2.2)
	Service Delivery	Establish the service delivery approach (SP 2.1)
	Configuration Management	Establish Configuration Management Records (SP 3.1)
Supply copies of documentation	Service System Development	

Table 3. Provision of SCM Information Services

The following are the descriptions of each CMMI practice for the provision of SCM Information Services configuration activity.

1. *Deploy Service System Components* installs the service systems in the delivery environment by ensuring that deployed components are kept under configuration control.
2. *Ensure Interface Compatibility* ensures interface compatibility between component and human interfaces.
3. *Validate the Service System* ensures that it is suitable for use in the target environment and fulfils the stakeholder requirements.
4. *Establish the service delivery approach* categorizes requests; assigns and transform responsibility for monitoring the status of request and actions in response to such requests of service delivery.

Table 5 shows the action items for the fourth activity in the framework which requires the verification of compliance to CM procedures that we have previously discussed. Procedure compliance verification is a straightforward activity during the development of traditional software systems. An auditor can observe participants in a software development activity and determine whether or not they are following the correct CM procedures. However, when SBAs are considered it may not be possible to observe all the stakeholders in the development cycle, particularly when a service is provided by a third party. In this case it may not be possible to ensure CM procedures are being adhered to unless there is an arrangement setup between the service provider and the service consumer. However in the case of SBAs it should always be possible for the application developers to manage and control the versions and configurations of applications that are in their control.

Action Items (from Galin)	Relevant CMMI – SVC Process Area(s)	Corresponding CMMI Practice(s)
Audit compliance to SCM procedures	Organizational Process Focus	Monitor the Implementation (SP 3.3)
Initiate updating and change of SCM procedures	Organizational Innovation and Deployment	Identify and Analyze Innovations (SP 1.1) Select Improvements for Deployment (SP 1.4)

Table 4. Verification of Compliance to SCM Procedures

The following are the descriptions of each CMMI practice for the Verification of Compliance to SCM Procedures configuration activity.

1. *Monitor the Implementation* ensures that organizational set of standard procedures and processes are implemented and deployed accordingly.
2. *Identify and Analyze Innovations* identifies innovations that can improve and update SCM and other relevant process areas.
3. *Select Improvements for Deployment* identifies updates and changes SCM procedures that will be based on organizational quality and process performance objectives.

Our research results in the following figure:

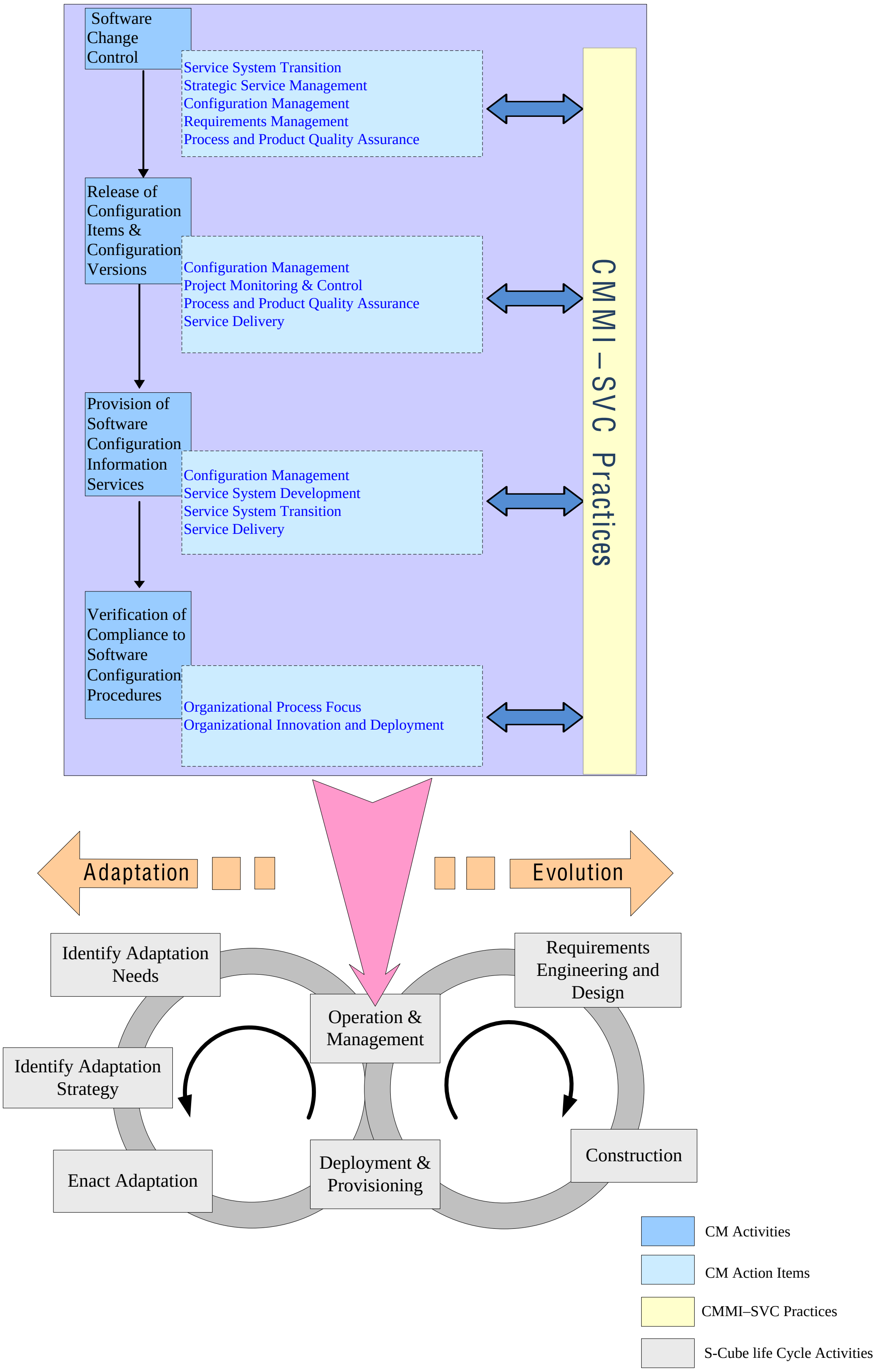

Software Change Control
Service System Transition
Strategic Service Management
Configuration Management
Requirements Management
Process and Product Quality Assurance
Release of Configuration Items & Configuration Versions
Configuration Management
Project Monitoring & Control
Process and Product Quality Assurance
Service Delivery
Provision of Software Configuration Information Services
Configuration Management
Service System Development
Service System Transition
Service Delivery
Verification of Compliance to Software Configuration Procedures
Organizational Process Focus
Organizational Innovation and Deployment
CMMI – SVC Practices
Adaptation
Evolution
Identify Adaptation Needs
Identify Adaptation Strategy
Enact Adaptation
Operation & Management
Deployment & Provisioning
Requirements Engineering and Design
Construction
CM Activities
CM Action Items
CMMI–SVC Practices
S-Cube life Cycle Activities

Fig 4. S-Cube Lifecycle with Detailed Operations and Management Phase

4 Case Study

4.1 Introduction

An S-Cube deliverable case study is presented [10] to support the implementation of our proposed framework. We discuss its execution in a case study focusing on a complex and geographically distributed supply chain in the automotive sector which has been offered by researchers of the companies 360Fresh and IBM [9]. In particular, we had to elicit real business goals and domain assumptions from the case study as they were not made explicit. Automobile Incorporation (Auto Inc), located in South East Asia, is a local branch of a large enterprise in the automobile industry in Europe. Its incorporation comprises of a regional headquarter in Singapore, a manufacturing factory in Vietnam, several regional distribution and logistics provider, and several warehouses located in different countries in South East Asia. Auto Inc sells automobile products to retail customers in the surrounding countries. Figure 3 illustrates the global business scope of the service network in our scenario. It highlights the main actors of the case study and the interactions, concerning both material and information flow, that occur among them. We used this figure as a basis to identify the basic business operations shown in the figure 5

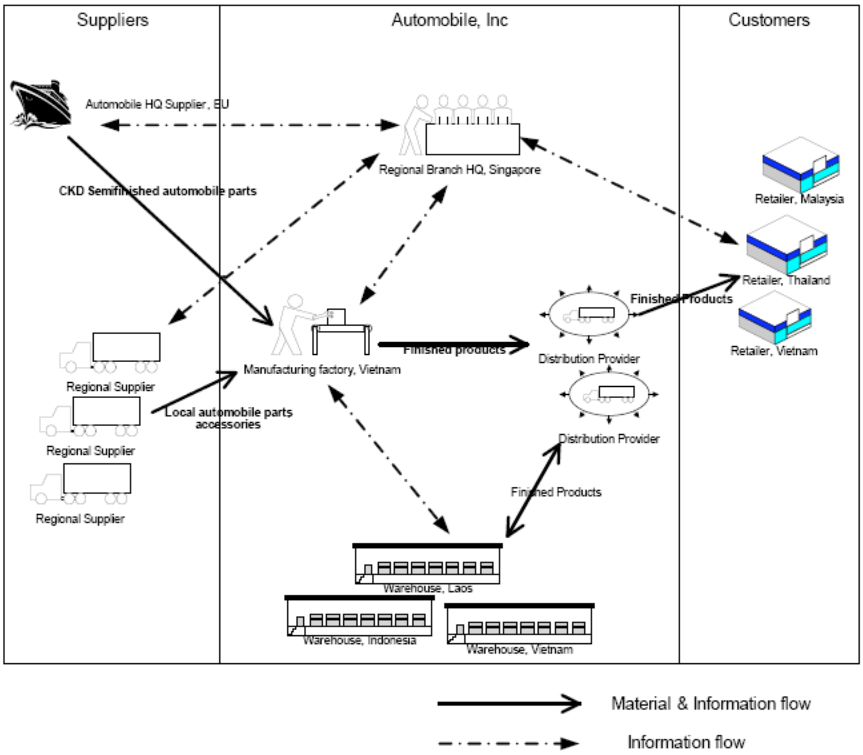


Fig 5. The business scope of our service network

4.2 Case Design

As the figure 5 shows, our service network consists of multiple warehouses, scattered across different geographical locations where finished products are stored from the manufacturing factory. Converting multiple warehouses into a single global warehouse can be a better option as it can benefit us in the following ways: 1) reduce the cost, and 2) minimize the overhead associated with management issues. The framework proposed in section 3 can benefit us in order to address issues associated with merging multiple warehouses into a single one. In Table 2, we identified set of CMMI practices which can help us to support the configuration management process; i.e. transition from a multiple into a single warehouse. We may select set of certain practices depending on the situation.

CMMI - SVC Practices	Description
• Analyze Issues (SP 2.1)	Analysis of issues associated with merger Counteractive measures to resolve the issues Supervise the procedure
• Take Corrective Actions (SP 2.2)	
• Manage Corrective Actions (SP 2.3)	
• Establish change management system (SP 1.2)	A change management system must be there to physically accommodate the change Categorize and recognize the products Maintain the record of finished and transferred products
• Identify Configuration Items (SP 1.1)	
• Establish Configuration Management Records (SP 3.1)	
• Prepare for Service System Operations (SP 2.2)	Set up for the operations once the change has been made Deal with incoming and outgoing orders/deliveries Maintain the record of finished and transferred products
• Receive and Process Service Requests (SP 3.1)	
• Establish Configuration Management Records (SP 3.1)	

Table 5. Case Design

5 Conclusions

Services have made the world more connected; allowing producers, consumers, and other human resources to communicate frequently across the globe. The service industry is a significant driver for the growth of worldwide economy. So guidance on improving service management procedure can serve as a key contributor to the customer satisfaction, performance, and profitability of the business. In this research, we have proposed a CMMI-SVC based framework to manage the configuration of service based applications. The hypothesis is supported by means of a case study to depict effectiveness of the approach that business can improve its CM process by means of our contribution. A special case with Service Oriented Architecture is that customer don’t see the change of services as long as Service Level Agreements are met. Yet, this is not how it is done nowadays and remains a future research issue for us. Another issue is that sometimes the providers of services in an SBA do not coincide with the SBA provider and they may be discovered dynamically during execution. We intend to use configuration information for the purpose of audit and for ensuring compliance between them.

6 Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube). It work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and grant PRTLI 4 to Lero Graduate School in Software Engineering.

7 References

[1] Enterprise Management Associates. “Configuresoft, Inc.’s Enterprise Configuration Manager: A Continuous Compliance Approach to Configuration Management,” 2006.

[2] Configuration Management Best-Practice Recommendations, Sun Services White Paper, May 2007

[3] The ITIL Toolkit – The ITIL Guide, ITIL - Office of Government Commerce, The Stationary Office, London, United Kingdom, Typical Page. [http://www.itil-toolkit.com/itil-guide.htm]

[4] Software Quality Management for SOA: Enterprise quality managers take the helm, white paper, Published by Hewlett-Packard

[5] Capability Maturity Model Integration for Services version 1.2, Technical Report CMU/SEI-2009-TR-001, ESC-TR-2009-001

[6] Capability Maturity Model Integration for Services version 1.2, Technical Report CMU/SEI-2009-TR-001, ESC-TR-2009-001

[7] Software Quality Assurance: From Theory to Implementation, Daniel Galin, Addison Wesley September 21, 2003

[8] Software Services and System Network, www.s-cube-network.eu

[9] J. Sairamesh, S. Zeng, B. J. Steele, and M. A. Cohen, “Dynamic service networks: Case study on supply-chain collaboration for early detection and recovery through information services,” technical report available from S-Cube on request.

[10] S-cube deliverables." [Online]. Available: http://www.s-cube-network.eu/results/deliverables

[11]A Guide to the Project Management Body of Knowledge (PMBOK Guide), Project Management Institute; 2000 ed edition

[12] Business component Factory, P. Herzum, O. Sims J. Wiley & Sons Inc., 2000.

[13] IEEE Std 828-2005 - IEEE Standard for Software Configuration Management Plans, IEEE Computer Society, 2005.

[14] A guide to Software Configuration Management, A.Leon, Artech House, Boston, MA, 1999.