ED 427 - École Doctorale Informatique de Paris Sud

Habilitation à Diriger des Recherches

Pascal POIZAT

Formal Model-Based Approaches for the Development of Composite Systems

Présentée et soutenue publiquement le 24 novembre 2011

Jury

Rapporteurs :
Prof. Mohand-Said Hacid, Université Claude Bernard - Lyon 1, France
Prof. Paola Inverardi, Università degli Studi dell'Aquila, Italy
Prof. Fabrice Kordon, Université Pierre et Marie Curie - Paris 6, France

Examinateurs :
Prof. Philippe Dague, Université Paris Sud, France
Prof. Marie-Claude Gaudel, Université Paris Sud, France
Prof. Ernesto Pimentel, Universidad de Málaga, Spain

Laboratoire de Recherche en Informatique, UMR 8623 CNRS
Université Paris Sud 11, Bâtiment 650, 91405 ORSAY CEDEX, France

Contents

CHAPTER 1

Introduction

My work addresses the application of formal methods to the development of software, that is *formal software engineering*. Software engineering is concerned about techniques and tools to build software pieces. More than providing a theoretical background for software engineering, formal methods support well-foundedness of the development process, including non ambiguous models and model transformations, and enable the automation of the whole, or parts of, the design and programming activities.

In this context, I have always been interested in the role played by *structuring* and *composition*. Blame it on the brick games of my early days, or on my enthusiastic discovery of object-oriented programming in the early 90's, I have always found interesting to build complex things out of simpler, smaller pieces, especially when one ends up *reusing pieces out of some initial blueprint*.

This document presents my recent work on structuring in software engineering. It does not try to give a comprehensive presentation of *all* my research works, nor to give *comprehensive technical details* that can be found in my papers[1].

In this chapter I first introduce the software architecture vision that is a consequence of structuring and composite software systems. Being agnostic to the software target technologies (*e.g.*, programming languages and middleware infrastructures) and to the formal methods been used in the development process, I will present first the software architecture concepts in a generic way, as done in Architectural Description Languages, before showing how these concepts instantiate on service architectures. I will then introduce composite system development processes and related issues. After stressing the recurrent principles in my work, I will finally present an overview of my contributions to these issues. This chapter also includes a short presentation of the formal models I use in my works. It can be skipped in a first reading.

[1]Most of my papers are available in my Web page. Further, five selected papers will be attached to this document.

1.1 Software Architectures

Motivation

A major issue one meets when designing a software system is its complexity. Complexity stems first from the size of the system, or from the different aspects that are to be taken into account when developing it. With recent outcomes both in industry (inter-enterprise applications) and for the end-users (social networks, electronic devices and increased connected mobility), software systems get more distributed, shared, and subject to on-demand construction and change than before. Accordingly software complexity increases since several software entities constituting the systems have to be taken into account. Further, these entities do not exist independently from one another. Their interactions are an important part of the problem as they can lead both to benefits (achieving some form of collaboration or cooperation) or to major drawbacks (deadlocks, starvation or incorrect multiple access to resources, etc.).

To solve out such complexity issues, approaches based on the decomposition of a system into sub-systems can be put into practice, whether is it for design, verification, or implementation. Behind the scene of such a decomposition, lies an important concept: *structuring*. It has been instantiated several times in the past years with approaches based on modules [Parnas, 1972], objects/classes [Dahl and Nygaard, 1966, Meyer, 1997], software components [Szyperski, 1998], services [Papazoglou and Georgakopoulos, 2003], aspects [Filman et al., 2005] or software product lines [Pohl et al., 2005].

Software components make the functionalities provided by some reusable entity explicit, as modules and objects/classes did. However, software components go further by making also the required functionalities explicit. Having both provided and required functionalities explicit supports the definition of software components independently from a specific usage context. This makes, in theory, software components more reusable than modules or objects/classes where requirements are hidden in the code. Services are the basic entities that underpin the development of Service-Oriented Architectures (SOA) and Service-Oriented Computing (SOC). But for differences in description languages and underlying middleware, services can be seen as a kind of software components. Aspect-orientation is a more asymmetric approach, where some first class entity, the base business model or code, is extended with the integration of one or several second class entities called aspects. Software product lines are another modern application of structuring, where products are the result of the integration of core elements and variation-related ones.

All the above-mentioned approaches lead to new software design processes with two distinct tasks[2]:

- *"designing in-the-small"*, the design, verification, and implementation or reuse of sub-systems satisfying a subset of the system's functionalities or corresponding to deployment units.

[2]The terms "designing in-the-large" and "designing in-the-small" are variations inspired from the terms "programming in-the-large" and "programming in-the-small" first introduced in [DeRemer and Kron, 1976].

- *"designing in-the-large"*, the structuring of the system as a set of sub-systems linked by dependency or interaction relations, a kind of structural blueprint of the system. In an analogy to building construction where buildings are constructed using an assembly plan for sub-parts (ground, walls, roof), such a blueprint can be referred to as the *architectural plan* or *software architecture* of the system, and the person in charge for its definition can be called a *software architect*.

Elements of an Architecture

In parallel with the development of structured design and programming techniques, the application of formal methods to the architectural vision of software has emerged with Architectural Description Languages (ADLs) [Medvidovic and Taylor, 2000], Coordination Models and Languages [Papadopoulos and Arbab, 1998], and more generally the application of formal methods to Component Based Software Engineering (see, *e.g.*, the FACS series of conferences) and Service Oriented Computing (see, *e.g.*, the ICSOC series of conferences). In the sequel, I adopt the ADL terminology (Fig. 1.1) to present architectural elements due to its technology agnosticism.

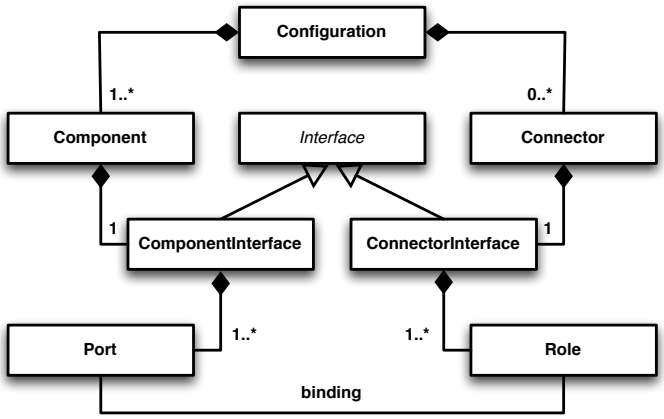

Figure 1.1: Simplified ADL metamodel (UML notation)

Components and Connectors

The basic elements of an architectural model are components and connectors. A *component* is the basic architectural entity that abstracts some information being available or some piece of code being reusable. To quote [Medvidovic and Taylor, 2000], *"A component is a unit of computation or a data store [...]. Components may be as small as a single procedure or as large as an entire application"*. The basic motivations that trigger the encapsulation of some information or piece of code as a component are the decomposition of a complex system into smaller sub-systems – the system components – and their interactions, or the desire to enable reuse, following the definition of Szyperski [Szyperski, 1998]: *"A software component is a unit of composition with contractually specified interfaces and explicit context*

dependencies only. A software component [...] is subject to composition by third-parties". The former case integrates perfectly in a top-down vision of the software engineering process (decompose to simplify), while the later case corresponds to a bottom-up vision (building value-added systems out of simpler ones).

A *connector* is the basic architectural entity that abstracts some type of interaction between components. A shared variable in a threaded Java program, a Unix pipe between two commands, a socket between a client and a server, a JDBC connection between a Java program and some database, an HTTP connection between a browser and a Web server, the SOAP connections between an orchestration and basic Web services, tuple spaces, etc. are all examples connectors in every-day life. It is a matter of discussion if there should be any (explicit) connectors at all. Some ADLs have no explicit connectors. It is particularly relevant when there are few kinds of interactions between components, *e.g.*, in Web service composition where SOAP over HTTP is the standard usage. Other ADLs advocate that if connectors are complex they should be defined as components, having again only simple bindings between "component" components and "connector" components, *e.g.*, a complex interaction between two Web services will lead to an orchestration component rather than a connector.

Interfaces

If we refer to the Oxford online dictionary [Oxford University Press, 2010], an interface is *"a point where two systems, subjects, organizations, etc. meet and interact"*. From a software point of view, the interface of an entity is both the *operational mechanism* through which this entity interacts with its environment (*e.g.*, a protocol information, an IP address, and a port, such as `http://127.0.0.1:8080/`), and the *contractual specification* of the entity functionalities (*e.g.*, *"this service provides you with the possibility to buy a trip package provided you give the departure and return dates/places, and for this it will require some plane ticket and hotel booking facility are available"*, Fig. 1.2).

As we saw before, an important benefit with reference to earlier approaches is that this specification explicits both the functionalities provided by the entity (here the trip package) and the ones it requires (here online services for plane and hotel booking). The first interpretation of interfaces is mandatory to enable implementability of software architectures. It gives an answer to the question *"Where can I access this entity?"*. However, ADL go further by using extensively the second interpretation and associated information, and giving an answer to the question *"What and how can this entity do something for me? – What and how should I do something for this entity?"*. Many ADLs refine interfaces as a set of interaction points that correspond to the first interpretation. An entity has then *one* interface, made up of *one or several* interaction points that can be either *provided* or *required*. In the component interfaces, these interaction points are called *ports*, while in the connector interfaces, they are called *roles* (they correspond to roles that components may play in an interaction).

As an alternative to entities having one interface with several interaction points, one may consider entities that have *several* interfaces and then do not decompose these into several interaction points, both notions coinciding. Web services for example take this

second interpretation. Let us take a composite Web service. It has different partners it will communicate with. These can correspond to end-users, programs, or other services that will fulfill some functionality for the composite. For each of these partners there is a port, called partner link, which is described using the WSDL language [W3C, 2001].

It is also usual that ports are detailed into sets of operations, the operation being the element being called, and the port being the place where the call is issued. Indeed, many different kinds of information can be associated to an interface (or respectively to ports and roles). This are often classified using four *interface description levels* [Canal et al., 2006] and respective *interface description languages* (IDL):

1. *Signature level.* This is the state-of-the-art of mainstream component and service middleware, *e.g.*, CORBA or SOA such as Web services. Interface descriptions at this level specify the methods or services that an entity offers, as in object IDLs, like CORBA-IDL, or the public interface of a Java class. Sometimes, they also describe its external dependencies, like in component IDLs like CCM-IDL for OMG's component model, or WSDL for Web Services. Typically, these interfaces specify the name of the service, the type of its arguments and return values, and possibly the exceptions raised, that is, the full signature of the entity behind the interface. This is typically what is described in Figure 1.2.

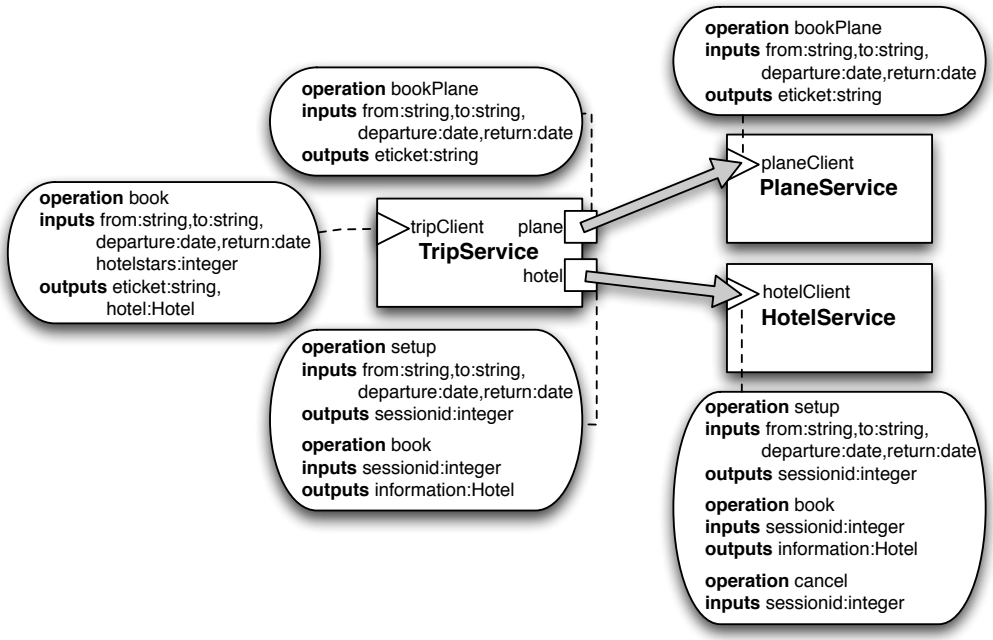

Figure 1.2: Signature interfaces (Acme ADL notation)

2. *Behavioral level.* Interfaces at this level specify the protocol describing the interactive behavior that a component follows, and possibly the behavior that it expects from its environment. Behavioral descriptions are required for entities

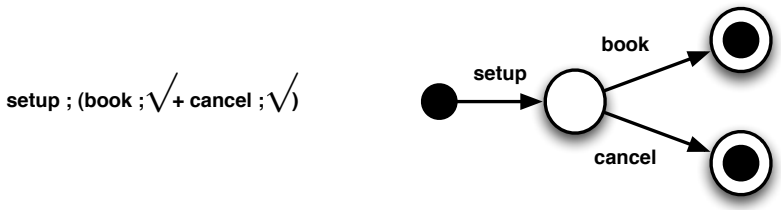

Figure 1.3: Behavioral interface (process algebraic and LTS notations)

with state, providing non-uniform services which are not available at any time, but that depend for example on the internal state of the entity or on the past history of interactions with their partner(s). This level is essential because the reuse and composition of components can lead to deadlocks or erroneous situations if one is not aware of their execution flows and does not take them into account while building the systems [Vallecillo et al., 2000]. Numerous proposals have been made for the extension of component interfaces with behavioral descriptions, *e.g.*, [Plášil and Visnovsky, 2002, Inverardi and Tivoli, 2003, Bracciali et al., 2005, Poizat and Royer, 2006, Barros et al., 2009]. It is also the case for (Web) services, *e.g.*, [Ben Mokhtar et al., 2007, Grigori et al., 2008, Bertoli et al., 2010]. Despite potential benefits (automated verification, adaptation and composition) many components do not present a behavioral interface. However, one may retrieve behavioral interfaces using process/protocol mining [Musaraj et al., 2010, van der Aalst, 2011], learning [Steffen et al., 2011], or testing [Bertolino et al., 2009]. ADLs with behavioral interfaces rely, for description and analysis, on formal behavioral model such as (labeled) transition systems (LTSs), Petri nets, or process algebras (see Sect. 1.5). Let us take again the trip service example. To be used, HotelService requires that its user, TripService, calls its three operations in a specific ordering. Such usage protocol can sometimes be inferred from the signature. Here, both the book and cancel operations require as input a session identifier that is provided by operation setup. However, in general behavioral signatures must be given. In our example, setup must be done first, and then either book or cancel may be called (Fig. 1.3, where ;, +, and $\surd$ denote respectively sequence, choice, and termination). This cannot be inferred from the signature.

3. *Semantic level.* This level describes what the entity actually does, not only the methods it offers, or the messages it exchanges. Such a kind of specification is particularly interesting for component mining, *e.g.*, automatic service discovery and selection [Benatallah et al., 2005b, Hacid et al., 2009]. In the field of Web services, this level of description is related to the Semantic Web, *e.g.*, using ontologies or description logics, and languages such as OWL/OWL-S or WSMO [Toörmä et al., 2008]. An alternative approach consists in using formal functional descriptions for specifying functionality, *e.g.*, preconditions and postconditions for the signature level operations.

4. *Non-functional level.* Numerous non-functional properties like security, reliability,

accuracy, cost, etc. are important to be taken into account when composing entities altogether. These properties are tackled by the non-functional interface description level. It is usually highly customizable, and the possible descriptions include mean values, standard deviations and a set of quantiles characterizing the distribution of any self-defined quality metric. This description level is of particular interest both for architectural analysis [Bernardo et al., 2002, Balsamo et al., 2004] and for component mining [Zeng et al., 2004, El Haddad et al., 2010].

If most works on software architectures associate with interfaces only a subpart of these four levels, the genericity and extensibility principles of interfaces promote that one can put in an interface description whatever is required to ease specification, analysis, and/or implementation. In my work, I have focused on the first three description levels.

Configurations

An (architectural) *configuration* is the description of an assembly of components (*i.e.*, a set of identified instances of component types) and connectors (*i.e.*, a set of identified instances of connector types). The relations between components (exchanging provided/required functionalities) is achieved through connectors and the binding of component ports to the different roles of the connectors.

Configurations represent an architectural viewpoint over a system. Different abstractions (*e.g.*, whether we are at design time, or at deployment time where deployment constraints are highly relevant) can lead to different configurations. Configurations enable the designer to check the system's architecture. Further, once a deployment scheme has been chosen (*e.g.*, if, for each component, an address is given) then automatic deployment of the system is possible.

An important benefit of configurations, when it comes to take the systems' complexity into account, is the possibility to build *composite components*, *i.e.*, to treat a configuration as a component and (re-)use it in further configurations. For this, it must be possible to relate the ports of the composite components to the ports of its sub-components. Composite components enable a divide-and-conquer strategy where, for example, a first decomposition level of a system can be made based on deployment issues (*e.g.*, having three high-level components in the system corresponding to three Web services deployed on three application servers), while in a second step each of these three components can be refined as a composite using a configuration made up of functional (business) sub-components (*e.g.*, implemented using Java classes). SOFA [Plášil and Visnovsky, 2002], Fractal [Barros et al., 2009], and the Service Component Architecture (SCA) [Open SOA, 2007] are examples of frameworks that are highly relying on composite components.

Composite components raise the *compositionality* issue. An architectural language or an architectural model is compositional if, not only is it possible to define composite components, but also the semantics of the configuration in the composite component can be related to the semantics of a component. This is the case with most formal ADLs with behavioral signatures: given the behaviors of all the components (and connectors, if not implicit) in a configuration, it is possible to retrieve the behavior for the configuration.

Compositionality raises the possibility of compositional verification of the architectures, *i.e.*, verify a composite component without computing the global behavior but rather taking into account only the sub-components' behaviors and their connections. Reasoning on composite components is not restricted to behavioral descriptions but it also applies to non-functional service descriptions, *e.g.*, to retrieve the value of quality of service attributes for composite components being given the value of these attributes for the sub-components.

1.2 Service Architectures

The ADL terminology presented above directly instantiates into concrete languages, models, and technologies such as the CORBA Component Model, the Service Component Architecture, or the Fractal Component Model. Service-Oriented Architectures (SOA) are becoming a key aspect for system agility and quickly developing new businesses. As core concepts of any SOA-based system, services have recently received significant interest. They can be used to support business-to-business, enterprise application integration, and collaborations within or between virtual enterprises/organizations. Services have also been used to encapsulate information in sensors and compose it. Web services constitute one of the most popular approach to implement the paradigm of service and are gaining industry acceptance and usage. They are also now studied in most university curriculums. For these reasons, it seemed to me natural to target Web services as the applicative domain related to my work on the application of formal methods to composite system development.

Table 1.1: Analogy between the ADL and WS terminology

ADL	WS
basic component	simple service (WSDL service)
configuration, composite component	composite service (orchestration, distributed orchestration, choreography)
signature interface	WSDL interface(s)
behavioral interface	conversation
semantic interface	semantic annotations in a WSDL file

A rapid analogy between the general ADL terminology and the Web service one can be done as shown in Table 1.1. I will present here the basic concepts of Web services required for the rest of the document. For a deeper introduction to the Web service domain, I refer to [El Haddad et al., 2009].

Simple services. *Simple Web services* advertise the functionalities they offer as a set of operations provided at a given port. This is defined in a Web Service Description Language (WSDL) interface [W3C, 2001]. Operation call, operation return, and faults are supported using messages, with operation input and output parameters being defined in the messages types. Operations may either be one way, with one input message, or two-way, with one input message and one output message.

Composite services: orchestration vs. choreography. Composite services may be implemented in a centralized way. This is *orchestration* (Fig. 1.4), where all messages pass

through the centralized composite service called *orchestrator*, and where other services are sub-services of the composition.

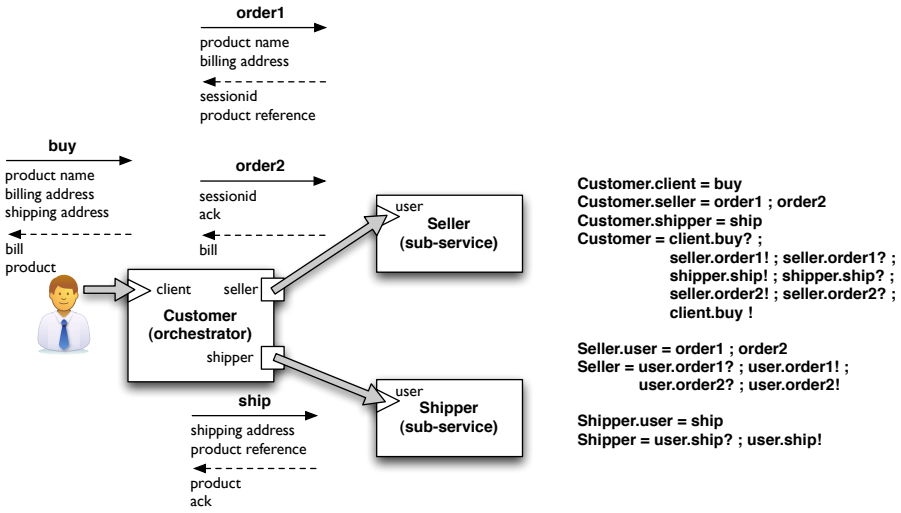

Figure 1.4: Orchestration and behaviors (Acme ADL + process algebraic notation)

Composite services may also be implemented in a distributed way. This is *choreography* (Fig. 1.5), where services are partners and communicate in a peer-to-peer fashion. The term choreography is also used to denote a kind of composition specification, and the distributed implementation may be called *distributed orchestration*. Indeed, choreographies are usually implemented using several orchestrators (one for each partner service).

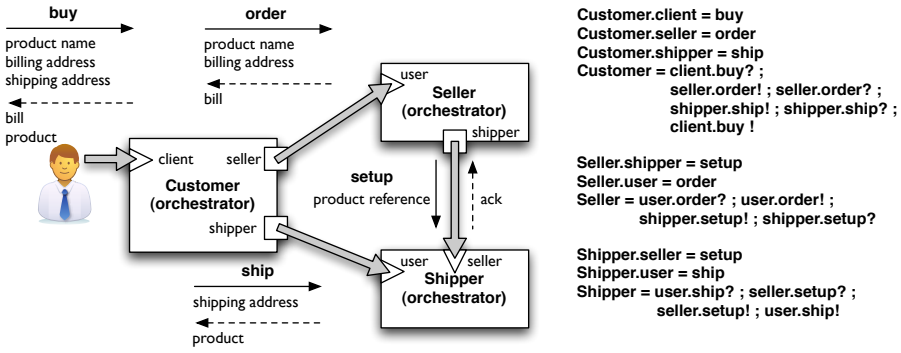

Figure 1.5: Choreography and behaviors (Acme ADL + process algebraic notation)

WS-BPEL, a composite service execution language. On top of a set of interfaces, an orchestrator is the implementation of a composition. Hence, an executable language should describe in which order the messages from/to different partner interfaces are received/sent. Many different languages can be used for this. All-purpose languages such as Java can be used provided a Web service stack is available (XML and RPC libraries for example). If the orchestrator is also to be a service (that is, its "input" ports are Web service ports) then an application server is required for deployment. However, the domain specific language paradigm prescribes that specific languages should be used for service orchestration. The widely accepted one is the Web Service Business Process Execution Language [OASIS, 2007] (WS-BPEL, or BPEL for short). One should note, still, that since BPEL execution engines can be heavyweight, some people have promoted the definition of lightweight orchestration languages such as JCWS [Boutrous-Saab et al., 2009] including the most useful BPEL constructs, *i.e.*, communication and workflow related ones, in pure Java.

The BPEL language is a workflow language based on an expressive set of activities enabling the orchestration of Web services. While the BPEL objective is to define executable orchestrations (*i.e.*, running on some execution engine), ABPEL has been defined to describe non-executable abstract processes such as behavioral interfaces. ABPEL is based on the same set of activities as BPEL. Here, I present the main ones. However, I advocate that this subset is the one that is to be found in (A)BPEL behavioral interfaces. A comprehensive presentation of BPEL can be found in [El Haddad et al., 2009].

Communication activities specify the communications between service partners. receive(o,$(v_1,\ldots,v_n)$) denotes the reception by a service of a message request for operation o with received data stored in variables v_i. reply(o,$(v_1,\ldots,v_n)$) denotes the corresponding reply by the service with output data taken from variables v_i. Accordingly, a service can call a partner service operation o using invoke(o,$(v_1,\ldots,v_n)$,$(v'_1,\ldots,v'_m)$) where sent data are stored in variables v_i and returned values to be stored in variables v'_j. When the operation is one-way (no return) one simply writes invoke(o,$(v_1,\ldots,v_n)$). Note that one only represents the operation name (o) in communication activities for simplicity. Taking into account both partner link p and operation name o, can be done through prefixing ($p.o$). *Assignment* (:=) supports data computation. exit denotes an empty (*e.g.*, terminated) process. In addition to these five *basic activities*, BPEL defines workflow-based *structuring activities*: sequence (sequence($P_1,\ldots,P_n$)), conditional activities (if(cond,P_{then},P_{else})), loops (while(cond,P)), and parallel flow (flow($\{P_i\}$)). BPEL also supports multiple event processing (pick($\{$onMessage $o_i,(v_{1_i},\ldots,v_{n_i})$: $P_i\}$[,onAlarm P_{alarm}])) where evolution of the process is triggered depending on either reception of a given message (onMessage $o_i,(v_{1_i},\ldots,v_{n_i}) : P_i$ yields P_i will be executed when message requests for operation o_i are received) or on a timeout (onAlarm P_{alarm}).

Orchestration specification. Abstracting away from the specification languages (see below for some examples), one can focus on the way orchestration specification is performed. If one wants to focus on the architecture of it, then an ADL can be used. If one want to focus on behavioral analysis, then there are different choices. The first one is to specify a *user interface only*, *i.e.*, the interaction protocol between the user (or any other system being the client of the orchestrator) and the orchestrator (*e.g.*, Customer.client in Fig. 1.4).

This is the way one end-user would define its requirements for an orchestration to be. Another solution is to specify *all interfaces of the orchestrator*, *i.e.*, for each sub-service, the interaction protocol between the orchestrator and this sub-service (*e.g.*, Customer.seller and Customer.shipper in Fig. 1.4). Such local protocols may be defined either at the operation level (as done, above) or at the message level. The third choice is to specify the whole orchestration protocol. However, in an implementation, the executable orchestration code combines messages being exchanged at all interfaces, hence protocols in this case are to be defined at the message level (*e.g.*, Customer in Fig. 1.4 where ? denotes a message reception and ! denotes a message sending).

There are numerous possible specification languages for the specification of an orchestration, including general purpose ones such as UML state transition and activity diagrams [OMG, 2005], workflow specific ones such as BPMN [OMG, 2011], and service specific ones such as ABPEL [OASIS, 2007], standardized by W3C, or SRML [Fiadeiro et al., 2006] and UML4SOA [Mayer et al., 2008], recent outcomes of the SENSORIA project. Such a freedom is important since there is no universal specification language. Yet, workflow languages are often advocated due to their relation with concepts in BPEL that is, after all, an executable workflow language.

In their work [van der Aalst et al., 2003], Van der Aalst and his colleagues have introduced twenty control flow patterns that are recurrent in workflow languages and specifications. This is only a part of all possible patterns (`http://www.workflowpatterns.com/`). It would not be feasible to present all of them here. However, it is possible to select a small subset of the control flow patterns that suits service orchestration and related implementation languages. We may define this subset as follows. Given a set of activity names A, a Workflow (WF) defined over A is a tuple $WF^A = (P, \rightarrow, N)$ [Kiepuszewski, 2003]. P is a set of process elements (or workflow nodes) which can be further divided into disjoint sets $P = P_A \cup P_{so} \cup P_{sa} \cup P_{jo} \cup P_{ja}$, where P_A are activities, P_{so} are XOR-Splits, P_{sa} are AND-splits, P_{jo} are XOR-Joins, and P_{ja} are AND-Joins. $\rightarrow \subseteq P \times P$ denotes the control flow between nodes. $N : P_A \rightarrow A$ is a function assigning activity names to activity nodes. Relation with other notations such as BPMN processes and UML activity diagrams is straightforward (Fig. 1.6).

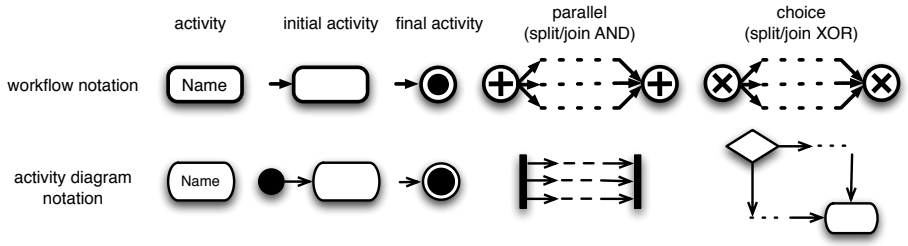

Figure 1.6: Workflow graphical notations (BPMN and UML notations)

Specification and verification (compatibility, architectural deadlock-freedom, temporal properties) both at the local and global level have been well studied in formal ADLs [Allen and Garlan, 1997, Bernardo and Inverardi, 2003] and more recently rephrased

in a service composition context [ter Beek et al., 2007]. Using them, one can check for example that Customer.seller and Seller.user in Figure 1.4 are compatible, or that the choreography (distributed orchestration) in Figure 1.5 is deadlock-free.

Choreography specification. Choreography as a specification language is the description from a global perspective of the interactions between roles played by peers (components, services, organizations, humans) in some collaboration. Several languages and notations do support choreography specification: BPMN 2.0, MSC, UML, WS-CDL to give some. Following [Decker et al., 2008], they can be classified using two dimensions. They can be abstract (*e.g.*, BPMN, MSC, UML) or concrete (*e.g.*, WS-CDL). Specification languages are used to describe *what* peers should (or should not) do in a collaboration, rather than *how* they should do it. Therefore, abstract choreography languages are better candidates for choreography specification. The second dimension is related to the underlying model. In *interconnected interface models* (*e.g.*, BPMN collaboration diagrams, MSC, UML sequence diagrams), conversations are defined at (each) peer level and interactions are defined by roughly connecting conversations. To the contrary, in *interaction models* (*e.g.*, BPMN 2.0 new choreography diagrams, UML collaboration diagrams), interactions between peers are the basic building blocks. From a designer perspective, interaction models better suit the needs of choreography specification due to their global perspective.

To understand the difference between a choreography specification and an orchestration one, let us take again our example in Figure 1.5. What we have here is three orchestration specifications (Customer, Seller, and Shipper). They could also be seen altogether as an interconnected interface model choreography specification, with connections being the gray arrows. An interaction model choreography specification would rather be described using any behavioral language (some being given above, but LTS or process algebras would work too) where atoms are interactions (that may thereafter be implemented using message exchanges or not). Such an atom can be denoted as $c^{[x,y]}$, where c is the interaction, and x and y are the interacting partners, with x being the initiator of the interaction. The corresponding specification could then be given as follows:

$$\begin{aligned} Choreo = \quad & order^{[Customer,Seller]}; order^{[Seller,Customer]}; \\ & ship^{[Customer,Shipper]}; setup^{[Seller,Shipper]}; \\ & setup^{[Shipper,Seller]}; ship^{[Shipper,Customer]} \end{aligned}$$

As for orchestration specifications, choreographies may be verified, *e.g.*, [Busi et al., 2006, Qiu et al., 2007, Foster et al., 2010, Basu and Bultan, 2011, Roohi and Salaün, 2011] for some recent references.

In the sequel, the composite systems development issues will be presented with a (Web) service perspective using the concepts that have been presented here.

1.3 Composite System Development

As mentioned in Section 1.1, software architectures enable both in-the-small and in-the-large development processes. Further, these processes are based on three distinct phases: development time, deployment time, and execution time (Fig. 1.7).

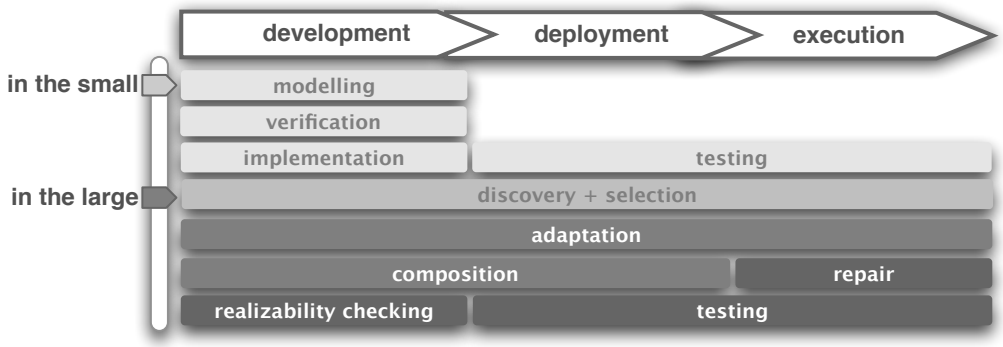

Figure 1.7: Activities in composite system development

In-the-small development. As far as in-the-small development is concerned, nothing changes from an usual process supported by formal methods (Fig. 1.7, upper part). Given some requirements, a model is designed. This model can be verified to check that it satisfies the requirements. For this, behavioral analysis techniques may be used, including model-checking. The model may then serve as a basis for implementation, which is further, verified using testing.

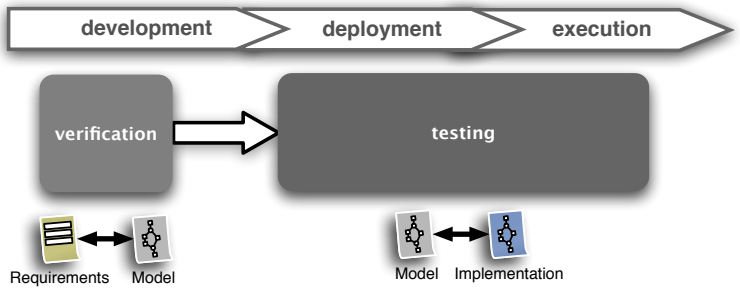

Figure 1.8: Verification (at design time) and testing

Design time verification and testing are complementary (Fig. 1.8) [Gaudel, 2011]. On the one hand, the former is required to ensure that one is not testing the implementation wrt. an erroneous model. On the other hand, excepted if one has access to the implementation code and a model of the running environment (in such a case, static analysis can be used), testing is required to check if the implementation conforms to its model. Moreover, due to their complexity, design time verification often abstracts away from some part of the systems. This can be for example time constraints (requiring more expressive formal models) or data being exchanged (raising state explosion issues). This over-approximation may lead to false negative results of verification. In such a case, testing may help in refining the abstraction [Beckman et al., 2010]. Testing is even more necessary with dynamic architectures such as services since the components to be reused are not always known in the earlier steps of composite systems development.

In-the-large development. Software architectures are most beneficial for in-the-large development, especially through partially or totally automated techniques for the composition, adaptation, testing, and repair of compositions (Fig. 1.7, lower part). Software architectures support both a bottom-up and a top-down development process (Fig. 1.9).

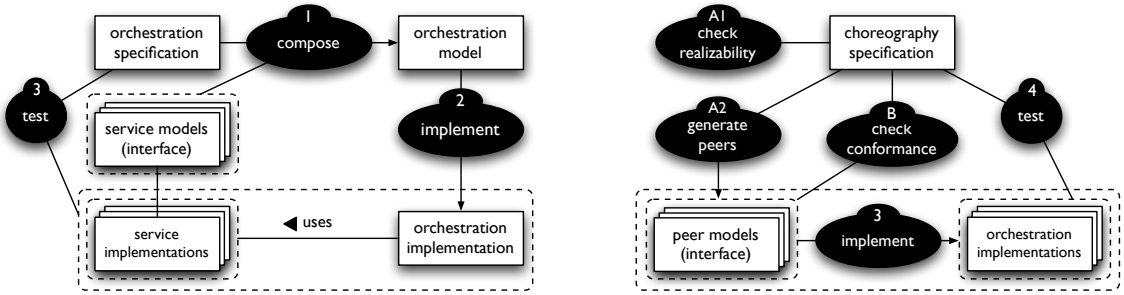

Figure 1.9: Bottom-up *vs.* top-down development

In a *bottom-up process* (Fig. 1.9, left), we begin with an orchestration specification and services to be reused, which may result from discovery (all services available in some repository that satisfy a given requirement are gathered) and selection (the best service(s) from the gathered ones are kept). In this document I will abstract from these two steps and suppose it as a preliminary. Recent works would be reusable here, either working on behavioral [Brogi and Corfini, 2007, Grigori et al., 2008], semantic [Benatallah et al., 2005b, Hacid et al., 2009], or non functional [Zeng et al., 2004, El Haddad et al., 2010] research criteria. The next step is to compose the services through first the retrieval of an orchestrator model (Fig. 1.9, left, 1) and second the implementation of this model (Fig. 1.9, left, 2). Finally the resulting orchestration is tested (Fig. 1.9, left, 3).

In a *top-down process* (Fig.1.9, right), we begin with a choreography specification, that is a global requirement. Using peer projection, local requirements can be generated [Qiu et al., 2007, Li et al., 2007] (Fig. 1.9, right, $A2$). An important issue here is realizability. It checks if the composition of such peer requirements obtained by projection from a choreography would expose or not exactly the same behaviors as the choreography (Fig. 1.9, right, $A1$).

Take for example choreographies $C_1 = {m_1}^{[P_1,P_2]}; {m_2}^{[P_3,P_4]}$ and $C_2 = {m_1}^{[P_1,P_2]}; {m_2}^{[P_3,P_1]}$. C_1 specifies that P_1 and P_2 must first interact on message m_1 (with P_1 being the initiator of the interaction) and then P_3 and P_4 will interact on message m_2 (with P_3 being the initiator). It is not realizable since there is no possibility for P_3 to know before sending m_2 that P_1 has sent (or P_2 has received) message m_1. In a distributed setting, implementing P_1 (resp. P_3) as sending m_1 (resp. m_2) we could have the case where m_2 is sent before m_1 which is forbidden by the choreography. Another issue is that realizability depends on the kind of communication. Take C_2 and suppose that P_3 sends m_2 and that P_1 is implemented as first sending m_1 and then receiving m_2. In a synchronous setting, the choreography is realizable since P_3 will be blocked upon sending m_2 until P_1 is ready to receive it. However, in an asynchronous setting, we would have the same problem as above.

Complementary to realizability is to check conformance: peer requirements are manually

written and are then compared to the choreography (Fig. 1.9, right, B). After checking, the peers may be either implemented directly (using an in-the-small process for them, Fig. 1.9, right, 3) or through reuse and composition (using the peer requirements as orchestration specifications and applying a bottom-up process for peer implementation, Fig. 1.9, left, 1–2). In both cases, testing is finally performed on the resulting distributed orchestration (Fig. 1.9, right, 4)

Composition. The composition (or coordination) of reusable software services enables, in theory, to gain time and money. In practice, if it is not automated, it is a complex and time-consuming activity: one must understand what a service does, how to reuse it and combine it with others, and check that the resulting assembly fulfills in practice the initial composition requirements. *Automated service composition* [Marconi and Pistore, 2009] tackles this issue with techniques for the discovery, selection, and assembly of services into composite services than can be deployed automatically. After being concerned about facilitating the enterprise software architects duties, automated service composition finds new applications with on-demand composition for the end-users and the "software as a service" cloud paradigm. The expressiveness and precision of the techniques is mainly related to the expressiveness and precision of the service descriptions, that is of their interfaces for which we have seen there are four different description levels.

A simple case is when the composition requirements are given as an abstract workflow (atoms are activities to be instantiated by service calls) and services have no conversation. If workflow and services are given at the same description level (operation or semantic capability for both), composition is mainly a matter of discovery and selection of the – locally or globally – best service for each activity in the workflow. However, an issue may arise. If the instantiated workflow, *i.e.*, the orchestration to-be, is correct from a control flow point of view (it naturally respects the abstract workflow behavior), it may not be correct with reference to a data flow point of view: some data being required for a service operation call may not be available in-time or as-is from previously called services. This is *horizontal mismatch*. Having requirements and services described at the same level is not always realistic, especially in case of on-demand composition from end-user requirements. Here the requirements would probably be described at an higher level (semantic data flow and capability control flow) while services would be described at a lower, implementation compatible, level (operations). This is *vertical mismatch*.

Further, services may present conversations. In such a case it is mandatory that the way the instantiated workflow uses the sub-services is *compatible* with their conversation. A typical example is given in Figure 1.10 where both services and composition requirements present conversations. Further, services are described at the operation level, and requirements at the capability level. Here we have three services, Conf to get conference information (infoC) and perform registration (registerC), Train to get a possible train ticket (infoT) and then either book (bookT) or cancel (cancelT) it, and Plane to get a possible flight ticket (infoP) and then either book (bookP) or cancel (cancelP) it. The three services use both basic information types (*e.g.*, place), and structured information types (*e.g.*, confinfo). The service conversations have been given in ABPEL but they could also have been given as LTS or using some process algebra. The end-user request, given as a workflow,

```
type confinfo : {n:confname, p:place, from:begdate, to:enddate}
Conf  =  while (true) { pick {
              onMessage(infoC,(topic)) : reply(infoC,(confinfo)) ; exit
              onMessage(registerC,(name)) : reply(registerC,(confid)) ; exit
          } } ; exit

type dates : {departure:begdate, return:enddate}
Train = receive(infoT,(dates,place)) ; reply(infoT,(tid,tproposal)) ;
          pick {
             onMessage(cancelT,(tid)) : exit
             onMessage(bookT,(tid,name)) : reply(bookT,(eticket)) ; exit
          } ; exit

type dates : {departure:begdate, return:enddate}
Plane = receive(infoP,(dates,place)) ; reply(infoP,(pid,pproposal)) ;
          pick {
             onMessage(cancelP,(pid)) : exit
             onMessage(bookP,(pid,name)) : reply(bookP,(eticket)) ; exit
          } ; exit
```

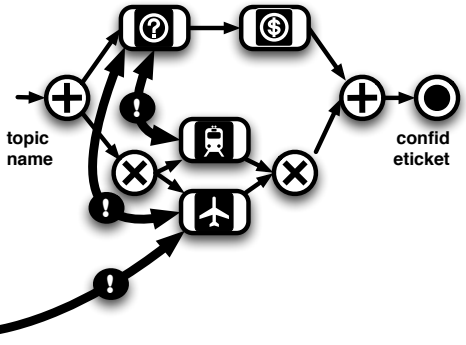

Figure 1.10: Composition in presence of conversations – inputs

expresses that conference related issues (first getting information, then registering) and transportation issues (either train or plane ticket reservation) can be done in parallel. Here we are in presence of horizontal mismatch: if getting conference information is performed at the same time than transportation booking, the date and place information will not be available. Further mismatch is due to the different structuring for these informations, *e.g.*, place and date information in Conf and Train (resp. Plane). We are also in presence of vertical mismatch: while services define operations (*e.g.*, bookP in Plane), the requirement uses capabilities (*e.g.*, plane booking).

```
Process-1 =
receive(USER,(topic,name)) ;
invoke(Conf.infoC,(topic),(confinfo)) ;
flow {
   invoke(Conf.registerC,(name),(confid)) ; exit,

   place := confinfo.p ;
   dates.departure := confinfo.from ;
   dates.return := confinfo.to ;
   invoke(Train.infoT,(dates,place),(tid,tproposal)) ;
   invoke(Train.bookT,(tid,name),(eticket)) ; exit
}
reply(USER,(confid,eticket) ; exit
```

```
Process-2 =
receive(USER,(topic,name)) ;
invoke(Conf.infoC,(topic),(confinfo)) ;
flow {
   invoke(Conf.registerC,(name),(confid)) ; exit,

   place := confinfo.p ;
   dates.departure := confinfo.from ;
   dates.return := confinfo.to ;
   invoke(Plane.infoP,(dates,place),(pid,pproposal)) ;
   invoke(Plane.bookP,(pid,name),(eticket)) ; exit
}
reply(USER,(confid,eticket) ; exit
```

Figure 1.11: Composition in presence of conversations – outputs

The outputs of an automated composition process should look like Figure 1.11. The two possible compositions respect the control flows expressed in services and in the requirements, while ensuring a consistent data flow from the user inputs to the required outputs (using service outputs also when needed) and solving mismatch out.

The existence of service and composition requirement conversations, together with horizontal and vertical mismatch require complex composition algorithms. A solution is to include adaptation features in these algorithms.

Adaptation. Services being developed by different third-parties, may present mismatch

at the different interface description levels that prevents them from being composed. Solving such mismatch out is the objective of software adaptation [Seguel et al., 2008], or adaptation for short. *Adaptation* consists in building automatically one (orchestration) or several (distributed orchestration) software pieces called adaptors restoring compatibility in a non intrusive way (Fig. 1.12, where bullets denote any kind -provided or required- of interface).

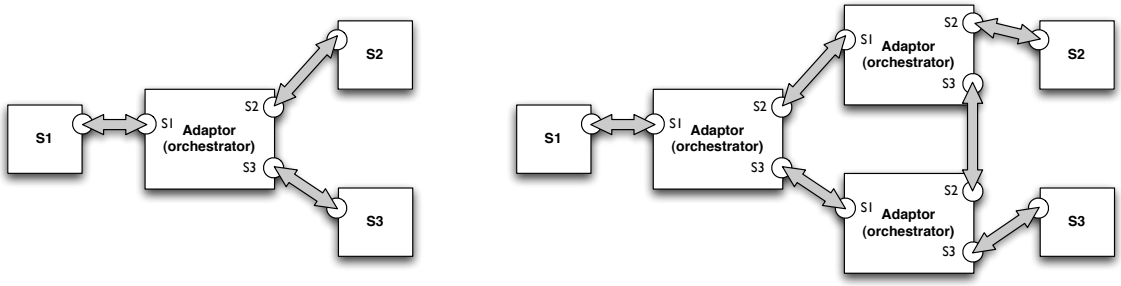

Figure 1.12: Centralized *vs.* distributed adaptation

Adaptation differentiates due to its dynamic, non intrusive, and automatic nature, with techniques such as evolution [Mens and Demeyer, 2008] that assumes that the reused pieces of code may be modified, or software product lines [Pohl et al., 2005] that assume that the possible use contexts or variations of a reusable component have been envisioned. Adaptation should also not be confused with adaptive middleware [Agha, 2002], that provides the technical means to adapt, and do not tackle the semi-automatic or automatic generation of the adaptor model(s) and code. Finally, adaptation is also sometimes compared in its objective with controller synthesis [Wonham and Ramadge, 1987, Ramadge and Wonham, 1989]. The main differences are that adaptors can perform message renaming, can reorder messages, and support the data associated to messages.

Let us take two simple services: a client C and a server S. Different kinds of behavioral mismatch may happen:

- Name mismatch (or 1-1 mismatch): an operation provided by S and an operation required by C have the same semantics but have different names, *e.g.*, two one-way operations, sendToPrinter and print.

- Unanticipated reception (or 1-0 mismatch): C tries to send a message to S while it is not used or required by it, *e.g.*, C, in non-connected mode, tries to send a login/password before each request, while S, in connected-mode, requires it only once.

- Generalized mismatch (or n-m mismatch): C and S use a different number of operations/messages for some task, *e.g.*, C wants to add two numbers x and y invoking in turn operations setX (to set x), setY (to set y) and add (to compute the sum) while S has a single operation, addition (to provide operands and compute the

sum at once). In this generalized mismatch, data may have to be aggregated (n-1) or split (1-n) amongst messages.

- Reordering: C and S have corresponding operations but different orderings, *e.g.*, C first sets up a file to operate on (setF) and then asks for an operation to be performed on it (perform), while S has first an operation for setting up the action (setA) and then an operation for doing the previously set action on a given file (run).

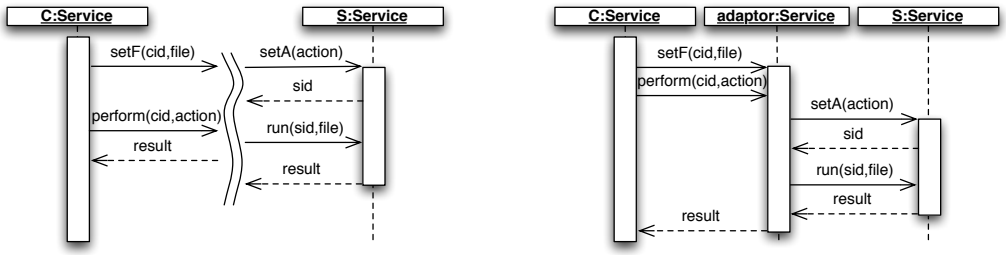

Figure 1.13: Mismatch between services (left) and adaptor (right)

These different kinds of mismatch are illustrated on a simple example, Figure 1.13, left, where the reordering case above is extended with new elements that will demonstrate how an adaptor works. C sends session identifiers (cid) with its messages to enable message follow-up. S also requires session identifiers to operate (sid), but these are generated by S and are different from the ones sent by C. As one can see from the figure, there is no chance that S can be used to fulfill the needs of C.

Adaptation techniques aim to automatically generate new components called *adaptors*, and may rely on an *adaptation contract*, which is an abstract description of how mismatches can be worked out. This is usually a morphism relating operations in the signatures interfaces of the adapted services. All the messages pass through the adaptor, which acts as an orchestrator and makes the involved services work correctly together by compensating mismatches. For instance, in the case of the name mismatch presented above, the adaptor would receive sendToPrinter from C and then would send print to S with the same data. If an unanticipated reception occurs, the solution is that the adaptor receives all client connection messages but only transmits the first one. In case of a generalized mismatch or a reordering issue, the adaptor would first receive from C all the necessary data and messages, and would interact with S only once all have been received. This is demonstrated on our example in Figure 1.13, right.

Adaptation is mainly about restoring compatibility, hence avoiding the deadlocks that are the consequence of mismatch between service protocols. For this, adaptation can be *restrictive*, forbidding interactions between services leading to deadlocks, and/or *generative*, renaming and reordering messages to avoid deadlocks.

Still, adaptation also fosters services composition in-the-large. Adaptation can be used in a bottom-up process (Fig. 1.14, left). Given services to be reused (1) and the composition requirement expressed as an additional service interface (2), adaptation generates, possibly

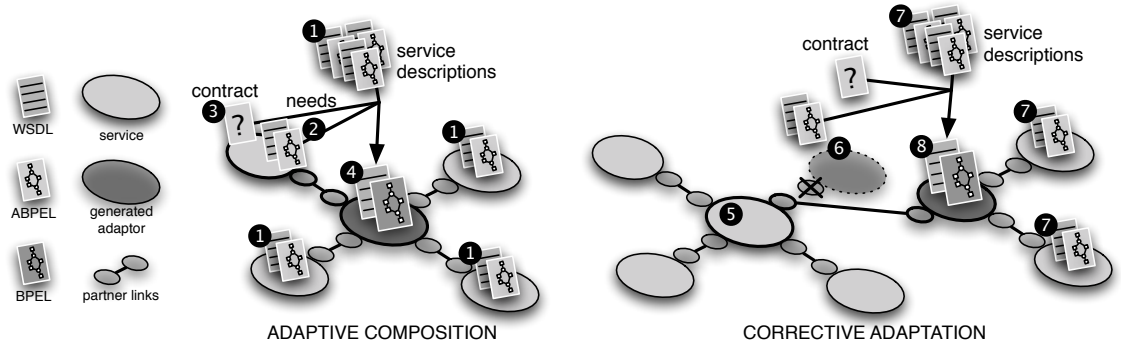

Figure 1.14: Using adaptation to empower service composition

with the help of an adaptation contract (3), a centralized adaptor under the form of an orchestrator (4). Adaptation can also be used when one service fails at run time or more simply when one wants to replace a sub-service in a composition (Fig. 1.14, right). Given the interface of the service to be replaced (6) and a set of replacement services (7), adaptation generates an adaptor between the client(s) of the replaced service and the replacement services (8).

Testing. The composition and adaptation processes are based on a key element: interfaces. It is therefore crucial to ensure that a service really does what it publicizes, and, as we have seen above, if a composition presents mismatch or not. Verification support this. Still, one has to face two important limits. First, the source code of services is not available (black-box assumption): only interfaces are published, and they usually describe much less than design artifacts. The second limit is related to the loosely-coupled and dynamic nature of service compositions. End-user requirements and services used to realize them may be known only at deployment or run time. Model-checking and behavioral equivalence techniques are of great interest during the design process to ensure for example that requirements correspond to what is really wanted, or that architectures known at design time are correct. However, with compositions being constructed at deployment time (end-user composition) and evolving dynamically, model based testing is required.

In *active testing*, the tester interacts with the implementation under test (IUT) by sending inputs (messages) and observing outputs (messages too). This method assumes a kind of controllability of the implementation through Points of Control and Observations (PCOs). Observing the outputs and comparing them to the expected ones, *i.e.*, those described by the specification, a verdict can be emitted. A *pass* verdict establishes the conformance of the implementation to its specification and a *fail* verdict yields the opposite. *Passive testing* is a software testing method that relies only on observations on the running IUT. In passive testing the tester does not send messages to the IUT. It only observes the exchange (sending and reception) of messages between the IUT and its partners, through Points of Observation (POs). These observations will be compared to the specification in order to emit a verdict. The term "passive" relates to the fact that the tests do not disturb the natural operation of the IUT, to the contrary of "active" testing. Passive testing is of

particular interest since one does not always have the ability to control an IUT, especially after deployment (Fig. 1.15).

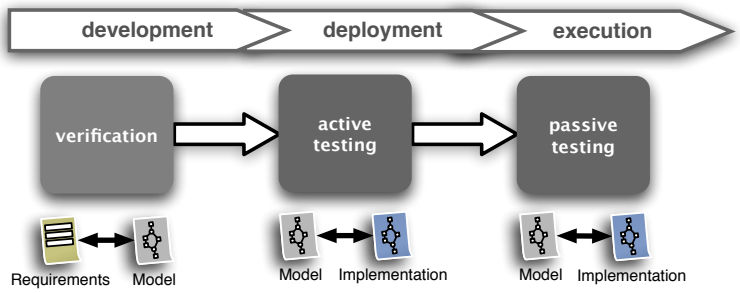

Figure 1.15: Verification (at design time), active and passive testing

Development as a continuous process. Service composition is today largely a static affair, as stated in [Papazoglou and van den Heuvel, 2007]. Due to user mobility, network problems, deprecation, etc. the services being used in a composition may become unusable, while other ones may appear. In a context of end-user composition, that is the automatic composition to fulfill on-demand user needs, these needs may also evolve. These issues are at the core of new highly dynamic and adaptive systems [Cheng et al., 2008, Bernardo and Issarny, 2011]. In this context, composition should be thought as a dynamic and continuous process (Fig. 1.16).

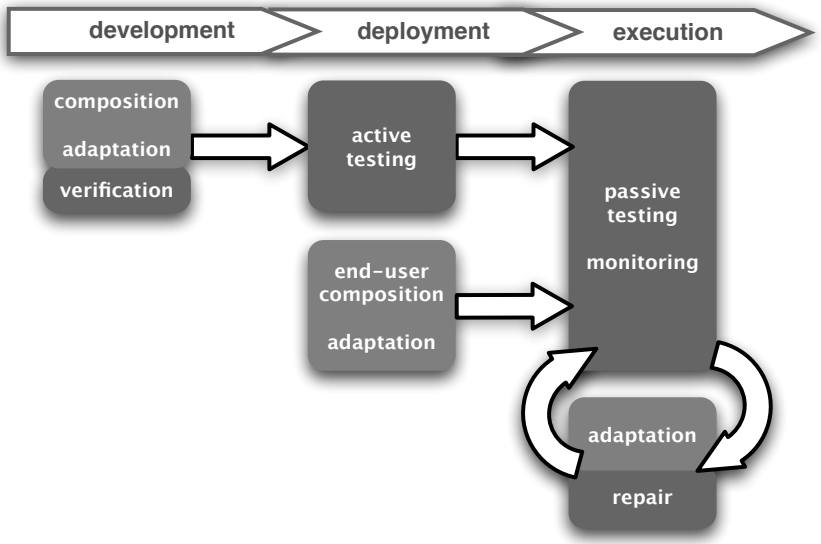

Figure 1.16: Activities in a continuous composition process

Here, a composition results either from an usual enterprise design (including composition, verification at design time, and active testing steps before publication) or as an end-user

request fulfillment. At run time, passive analysis techniques are to be used to check the correctness of the composition with reference to its dynamic environment. This can be done using monitoring or testing. In case of a problem, yielding a broken composition, corrective measures should be taken (using online adaptation and repair) to yield a new composition.

In my work I propose formal techniques for the composition, adaptation, testing (both active and passive), and repair of composite systems. As such, I try to enable this new vision of composition as a continuous process.

1.4 General Principles

For proposing solutions to the software engineering issues presented in the previous section, I defend and I have followed some interrelated general principles:

- (P1) importance of formal models
- (P2) absence of universal formal model.
- (P3) use of a layered formal model transformation approach.
- (P4) importance of behavioral interfaces.
- (P5) automation of the solutions and tool support.
- (P6) connection with the applicative domain.

(P1) importance of formal models. Formal methods promote the development of well-founded solutions and their automation. Further, using formal methods for software engineering issues enables one to reuse not only the related theory, but also to gain tool-support. The Model Driven Engineering (MDE) paradigm, that has now emerged in software engineering, is also calling for formal models as demonstrated for example in [Combemale et al., 2009] where the use of bisimulation theory enables to ensure correctness of model transformations. Still, the use of formal models in MDE and the seamless integration of activities supported by formal methods in the software development process could be further developed, as proposed, *e.g.*, in [Kordon et al., 2008] for verification activities, switching from MDE to Verification Driven Engineering (VDE).

(P2) absence of universal formal model. It is known that there is no universal language or model. It would be so expressive that its complexity would prevent developing any practical solution on it. This complexity would anyway also prevent its use by non specialists, *e.g.*, non academic software architects or end-users. Rather, specific languages or models should be used, *i.e.*, defined or reused, for specific applicative domains (here, composite systems) and specific issues (composition, adaptation, verification).

(P3) use of a layered formal model transformation approach. Having languages and models being specific for applicative domains and for issues yields not only domain specific languages (and models) but also issue specific working languages (and models).

This promotes the development of a layered approach based on different languages and models, and on model transformations (Fig. 1.17).

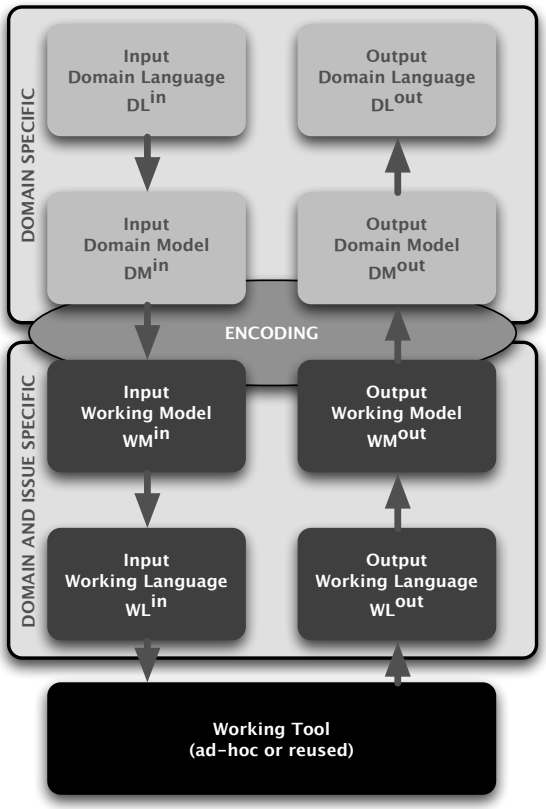

Figure 1.17: Layered formal model transformation approach

The left hand part of the figure is relative to the issue and the right hand part to the solution (if it exists). Taking domain specific inputs (*e.g.*, a service composition requirement language), one retrieves models for them (*e.g.*, one of those presented in the next section). This model is transformed using an encoding into another model that is suitable for an issue of interest (*e.g.*, some planning structure for automated composition). This model is then dump in a language that is understood by some ad-hoc or reused tool. Once the tool has run, one proceeds the other way round, reading the tool outputs, transforming them in some domain specific model (*e.g.*, an orchestration formal model) and finally in some domain specific language (*e.g.*, the BPEL orchestration language). This architecture promotes the reuse of domain language from/to domain model transformations (*e.g.*, (A)BPEL from/to STS) in different issue contexts. Still, this does not prevent that domain and working models coincide. This architecture shares some of its objectives with VDE [Kordon et al., 2008], having formal activities at the core of the software development process, with different tailored models and techniques to be used for different properties and at different steps of this process.

(P4) importance of behavioral interfaces. We have seen that components (including services) could be described at four different interface description languages. Industrial models and languages by using signature level interfaces and middleware are able to solve technical issues such as component heterogeneity. However, the behavioral description level is essential because even if components match from a signature point of view, their combination can lead to deadlocks or erroneous situations if one is not aware of their execution flows and does not take them into account while building the systems.

(P5) automation of the solutions and tool support. The manual application of engineering principles is tedious and error prone. In order to be applicable, the proposed solutions should go beyond proposing notations, patterns, or structuring rules to be applied by the software architect. These solutions should be automated, which, further, paves the way for on-demand composition by end-users and self-corrective systems. Automation includes the development of prototype tools in order to validate empirically the solutions.

(P6) connection with the applicative domain. New applicative domains such as SOA promote initially approaches that address domain specific issues in a very abstract and theoretical way. This is valuable to ease the application of principle P1. However, models are also there to model and address questions on some reality. Proposing solutions for the development of composite systems without connecting them to any real-life composition framework would be somehow stopping half-way. First, software engineering is about software. One cannot therefore develop a software engineering technique without addressing the relation with software in some sense, *e.g.*, relating standard requirement languages with formal models, orchestration models with execution languages, or passing test cases using SOAP rather than just generating abstract test cases. Second, practical implementations bring additional issues, *e.g.*, the role of BPEL engines in the execution semantics of an orchestration, that should be taken into account to assess proposed formal solutions.

1.5 Preliminary on Formal Models

In my work, I have used different formal models. In this section I briefly present them. This can be skipped in a first reading and serve as a background complement for concepts and techniques used in the contributions presented in Chapters 2 and 3.

Events. *Events* are the base building block of a behavioral description. Therefore, the first step in developing a behavioral formal approach for software engineering is to define the set of events. As far as software architectures and their service instantiation presented above are concerned, events fall in one of the following categories:

- *interaction events* are relative to the interaction between components in a configuration (or composite component). For a message based implementation (Web services), these events correspond to the sending and the receiving of a message m (input or output) relative to some operation o in some partnerlink p. Hence, such events are denoted with $Ev^{\text{comm}} = Ev^{\text{in}} \cup Ev^{\text{out}}$, where events in Ev^{in} (inputs) are of the form $p.o.m?$ and events in Ev^{out} (outputs) are of the form $p.o.m!$. When the relation

between operations and messages is clear from the context (it can be inferred from the signature/WSDL file) one can use only $p.o?$ and $p.o!$, or $m?$ and $m!$.

- the *internal event*, τ, is used to denote internal actions of components (*e.g.*, assignment) or of composites (*e.g.*, internal interactions between sub-components not being observable from outside).
- the *termination event*, $\surd$, is used to denote the correct termination of a process and differentiate from deadlock situations.
- the *time passing event*, χ, is used to model a discrete version of time (time constraints of a Web service are generally soft, thus such a discretization of time is a valid abstraction [Haddad et al., 2004a]).

The set of events used in behavioral formal models may then be defined as

$$Ev = Ev^{\text{comm}} \cup \{\tau, \surd, \chi\}$$

Depending on the abstraction level, *e.g.*, mainly in our context whether data associated to messages are taken into account or not, interaction events may support additional variables (*e.g.*, for message formal parts in message reception, $p.o.m?x, y$) or values (*e.g.*, for message effective values in message sending, $p.o.m!"Hello", 123$). For some given formal model representing a component or a behavioral interface, its set of events, included in Ev, is also called *alphabet*.

Trace Sets (TSet). A trace is used to denote a sequence of events, and a trace set is simply a set of traces. Without entering the battle between linear *vs.* branching models and logics [Vardi, 2001], trace sets are used mainly in relation with an implementation, in which a given sequence of events has happened (hence where branching is not needed), *i.e.*, logs.

Definition 1.5.1 (Trace)**.** *Given an alphabet A, a trace t is a word on A, i.e., $t \in A*$.*

A trace on A is usually denoted $\langle a_1, \ldots, a_i, \ldots, a_n \rangle$, with $a_i \in A$ for each a_i, and $\langle\rangle$ denotes the empty trace. One is often interested in prefix-closed trace sets, which are sets of traces T such that $\forall \langle a_1, \ldots, a_i, \ldots, a_{n-1}, a_n \rangle \in T \, \langle a_1, \ldots, a_{n-1} \rangle \in T$. Operator . (resp. $\frown$) denotes event/trace (resp. trace/trace) concatenation.

Labeled Transition Systems (LTSs). State transition models have the benefit to combine formalism and user-friendliness. This is the reason why that are often used in industry, with, *e.g.*, UML state diagrams. Labeled Transition System (LTSs) are a simpler version of these, without expressive structuring mechanisms such a hierarchical or parallel states. LTSs can be found under different names with small variations, mainly in the supported events and their structuring (or not) as inputs and outputs, and in the product definitions. The LTSs we use originate from the seminal work by Arnold [Arnold, 1994]. A major interest of LTSs is tool support. Another interest of LTSs is that they can be retrieved using the operational semantics of various other formal models, such as process algebras, event structures, and Petri nets.

Definition 1.5.2 (Labeled Transition System)**.** *A Labeled Transition System is a tuple* (A, S, I, F, T) *where* A *is an alphabet,* S *is the set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are the final states, and* $T \subseteq S \times A \times S$ *is the transition function.*

A transition (s, a, s') in T is also denoted $s \xrightarrow{a} s'$. We define $deg(s) = |\{s \xrightarrow{a} s' \in T\}|$, for any s in S. A deadlock is a state s in S such that there is no transition $s \xrightarrow{a} s'$ in T ($deg(s) = 0$). Final states have the same objective as termination events, *i.e.*, make a difference between correct termination (being in a final state) and deadlock (being in a state without outgoing transitions). In presence of termination events in the alphabet, F is defined as the set of states $\{s' \in S \mid \exists s \xrightarrow{\surd} s' \in T\}$.

The main operations used with LTSs are projection, hiding, products, behavioral reductions, and behavioral equivalences. Let $L = (A, S, I, F, T)$ be an LTS. Its *projection* on an alphabet A' is the LTS L in which all labels that are not in A' have been replaced by τ. *Hiding* is the complementary operation, *i.e.*, the hiding of an alphabet A' in L is the LTS L in which all labels that are in A' have been replaced by τ. In both cases, the resulting LTS may then be reduced using a τ reduction (see below). The principle of an LTS product (we give here a generalized binary version of it inspired by the vector product [Arnold, 1994]) is to retrieve from two LTSs $L_1 = (A_1, S_1, I_1, F_1, T_1)$ and $L_2 = (A_2, S_2, I_2, F_2, T_2)$ an LTS $L = L_1 \times_X L_2 = (A, S, I, F, T)$ with $X \subseteq Ev \times Ev$ such that:

- $A \subseteq A_1 \times A_2$, $S \subseteq S_1 \times S_2$, $I = (I_1, I_2)$, $F \subseteq F_1 \times F_2$,
- $(s_1, s_2) \xrightarrow{(a_1,a_2)} (s'_1, s'_2) \in T$ iff $s_1 \xrightarrow{a_1} s'_1 \in T_1$, $s_2 \xrightarrow{a_2} s'_2 \in T_2$, and $(a_1, a_2) \in X$.

This product can be easily extended to n LTSs. There are several LTS products defined in the literature. The more used ones are the synchronous and the asynchronous products. Let $\overline{a}$ denote the complementary event of a, which has the property that $\overline{\overline{a}} = a$ and that, depending on the LTS labels, may be defined as $\overline{a} = a$ (simple events without directions), $\overline{p.o.m?} = p.o.m!$ (differentiation between input and output events), or even using more complex relations (based on synchronous vectors for example). We also have $\overline{A} = \{\overline{a} \mid a \in A\}$. The *synchronous product* corresponds to the case where $X = (A_1 \times \overline{A_1}) \cup (\overline{A_2} \times A_2)$. The *asynchronous product* corresponds to the case where $X = (A_1 \times \epsilon) \times (\epsilon \times A_2)$, with ϵ denoting a do-nothing event and the LTSs being previously modified to add a transition $s \xrightarrow{\epsilon} s'$ for each state $s \in S$. One may also define a loose version of the synchronous product that requires synchronization on common events only.

Synchronous products are particularly interesting to retrieve the semantics of interacting components and check for deadlock issues and incompatible protocols. Asynchronous products are particularly interesting to retrieve the semantics of non-interacting components, for example when one wants to use them with reference to a composition.

Behavioral equivalences (resp. preorders) between LTSs are numerous, see, *e.g.*, some of them in [van Glabbeek, 2001]. Given some equivalence $\mathcal{R}$, a $\mathcal{R}$-reduction is to retrieve from an LTS L a smaller LTS L' (usually the smallest one) such that $L'\mathcal{R}L$. Among the equivalences and reductions being used on LTS, τ reductions are particularly interesting since they enable to remove internal events in models while keeping the same behavioral semantics, up to the equivalence of course.

Products are used to retrieve global models (composition behavior) from local ones (sub-component behaviors). Conversely, projections are used to retrieve local models (component-level requirements) from global ones (composite-level requirements). Hiding is used to hide in a composition events relative to internal interactions between the sub-compositions. Using product, hiding, and reduction, one achieves compositionality with LTSs: a composite component made up of several interacting sub-components can be reduced to a semantically equivalent simple component.

Process Algebras (PAs). Numerous process algebras (and calculi) have been introduced in the literature [Baeten, 2005], *e.g.*, simple ones (ACP, CCS, CSP) or extensions with value passing (LOTOS, μCRL, PSF), time (numerous CCS extensions), mobility (π-calculus), stochastic information (EMPA, PEPA), etc. With reference to LTSs, their first interest is their conciseness and their more declarative style: while $a||b||c$ is quite short to denote the interleaving of the three events a, b, and c, an equivalent LTS would contain them explicitly, with 8 states and 12 transitions.

A process algebra is given syntactically as a set of atoms (in our context, events) and structuring operators, and semantically as a set of structural operational semantic rules and / or calculus axioms. A typical and simple example would be as in Figure 1.18 with the syntax on the left and semantic rules for ; and for + on the right.

$$
\begin{array}{lcll}
A & ::= & pl.o.m? & \textit{(reception of a message)} \\
 & | & pl.o.m! & \textit{(sending of a message)} \\
P & ::= & \surd & \textit{(termination)} \\
 & | & A;P & \textit{(sequence)} \\
 & | & P_1 + P_2 & \textit{(choice)} \\
 & | & P_1||P_2 & \textit{(synchronous product)} \\
 & | & P_1|||P_2 & \textit{(asynchronous product)}
\end{array}
\qquad
\frac{}{a;P \xrightarrow{a} P}
\qquad
\frac{P_1 \xrightarrow{a_1} P_1'}{P_1+P_2 \xrightarrow{a_1} P_1'}
\qquad
\frac{P_2 \xrightarrow{a_2} P_2'}{P_1+P_2 \xrightarrow{a_2} P_2'}
$$

Figure 1.18: A simple process algebra (syntax and part of the semantics)

Given a process algebraic description P, one may work at the semantic level, with the process algebra operational semantic rules to retrieve an LTS model for P (using the given rules one can relate the process and the LTS in Fig. 1.3) and then using behavioral equivalence and model-checking tools. A second interest of process algebras is that an alternative (if we suppose the calculus is correct and complete) is to work at the syntactic level, using the calculus axioms and rewriting tools or theorem provers.

Symbolic Transition Systems (STSs). STSs have been introduced to give a symbolic semantics to value-passing process algebras in [Hennessy and Lin, 1995]. STSs have later been employed, under different forms and names [Poizat and Royer, 2006], to associate a behavior with a specification of data types that is used to evaluate guards, actions and sent values in extended transitions.

Definition 1.5.3 (Symbolic Transition System)**.** *Given a set of value extended events Ev,*

a Symbolic Transition System is a tuple $(\mathcal{D}, \mathcal{V}, S, s_0, T)$ *where* $\mathcal{D}$ *is a set of domains,* $\mathcal{V}$ *is a set of variables with domain in* $\mathcal{D}$*,* S *is a non empty set of states,* $s_0 \in S$ *is the initial state, and* T *is the transition relation,* $T \subseteq S \times \mathcal{T}_{\mathcal{D}^{\text{Bool}},\mathcal{V}} \times Ev \times seq(Act) \times S$*, with* $\mathcal{T}_{\mathcal{D}^{\text{Bool}},V}$ *denoting Boolean terms, and* $seq(Act)$ *denoting sequences of actions, i.e., assignments from* $\mathcal{T}_{\mathcal{D},\mathcal{V}}$ *(terms) to* $\mathcal{V}$ *(variables).*

The term 'symbolic' is not related to symbolic model-checking. It means supporting the symbolic treatment of data variables, as opposed to enumeration over the domain of these variables which is the case when one retrieves an LTS from a data enriched model. STSs share with transition systems their graphical form. With reference to LTSs, transitions in STSs may have guards (boolean expressions enabling the transition), reception variables, and actions. STSs share operation kinds with LTSs, with *e.g.*, symbolic bisimulation, symbolic products and semantics through symbolic execution.

Symbolic Execution Trees (SET). Symbolic execution [King, 1976] (SE) is a program analysis technique that has been originally proposed to overcome the state explosion problem when verifying programs with variables. SE represents values of the variables using symbolic values instead of concrete data [Khurshid et al., 2003]. Consequently, SE is able to deal with constraints over symbolic values, and program output values are expressed as a function over the symbolic input values. The SE of a program is represented by a symbolic execution tree (SET).

Definition 1.5.4 (Symbolic Execution Tree)**.** *Given a program with identified steps* S *and operating over a set of variables* $\mathcal{V}$*, a Symbolic Execution Tree is a tree* $(\mathcal{N}, \mathcal{E})$ *where* $\mathcal{N}$ *are (symbolic) nodes and* $\mathcal{E} \subseteq \mathcal{N} \times Ev \times \mathcal{N}$ *are the edges. The nodes in* $\mathcal{N}$ *are tuples* $\eta_i = (s, \pi, \sigma)$ *where* $s \in S$ *is the program counter,* σ *is a map* $\mathcal{V} \to \mathcal{V}_{\text{symb}}$ *where* $\mathcal{V}_{\text{symb}}$ *is a set of (symbolic) variables disjoint from* $\mathcal{V}$*, and* π*, the path condition (PC), is a boolean formula with variables in* $\mathcal{V}_{\text{symb}}$*. Events,* Ev*, are extended with symbolic variables (*$\mathcal{V}_{\text{symb}}$*).*

The PC in the nodes of an SET accumulate constraints that the symbolic variables must fulfill in order to follow a given path in the program. SE has been applied to the verification of interacting/reactive systems, including testing where it can be used to generate symbolic test cases (STC) corresponding to traces in the SET.

An example is given in Figure 1.19. While the SET has only 10 states and 9 transitions, a corresponding non symbolic model (LTS) would be unbounded due to the reception of natural numbers in the first transition. Bounding them to [0..256], would yield an LTS with 66,565 states and 132,612 transitions.

Labeled Event Systems (LES). Event structures [Boudol and Castellani, 1989] are important models as far as true concurrency semantics is concerned. Other models for this indeed exists [Boudol et al., 2008] but event structures are interesting due to relations that have been studied with process algebras [Boudol and Castellani, 1989] and Petri nets [Winskel, 1986].

Definition 1.5.5 (Labelled Event Structure)**.** *Given a set of labels* A*, a Labeled Event Structure is a structure* $S = (E, \prec, \#, l)$ *where* E *is the set of events,* $\prec \subseteq E \times E$ *is an*

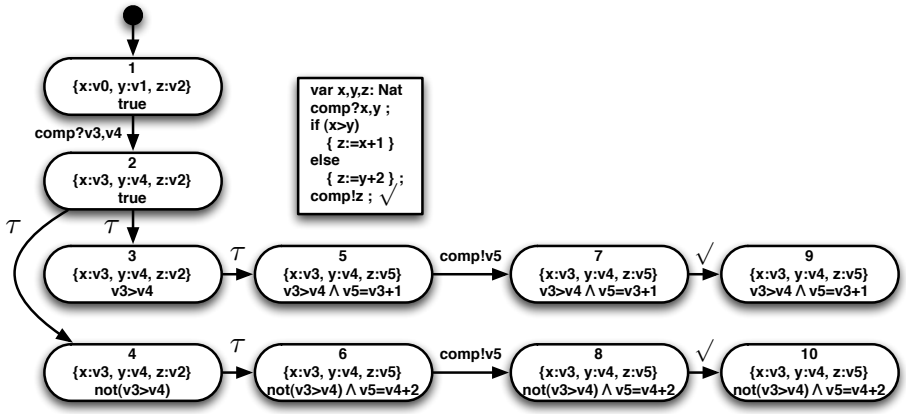

Figure 1.19: Symbolic Execution Tree

irreflexive relation, the flow relation, $\# \subseteq E \times E$ *is a symmetric relation, the conflict relation, and* $l : E \to A$ *is the labeling function.*

In an LES, causality (*e.g.*, coming from sequences in component protocols) can be expressed with $\prec$, exclusive events (*e.g.*, coming from alternatives in component protocols) by $\#$, and concurrency (*e.g.*, coming from activities performed in parallel in component protocols) by two events that are related neither by $\prec$ nor by $\#$. This makes LESs a very interesting model when it comes to work with workflow languages.

The semantics of an LES [Boudol and Castellani, 1989] is given through the definition of configurations (a subset of non conflicting events of E closed by causality) and of computations (a partially ordered set made up of a configuration and its causality relations). This semantics of an LES corresponds to an LTS where states are LES and labels are computations, in the same style as the semantics of a PA was an LTS where states are processes and labels are events. A comparison between LTS and LES on a simple example can be made looking at Figures 2.7 (page 49) and 2.9 (page 53).

Petri Nets (PNs). Petri nets [Reisig, 1985] are an expressive formal model for distributed systems modeling. Due to their graphical notation, they are accepted in industry, further they can be easily related to standard notations such as UML activity diagrams and BPMN.

Definition 1.5.6 (Petri Net)**.** *A Petri net is a tuple* (S, T, F)*, where* S *is the finite set of places,* T *is the finite set of transitions, and* $F \subseteq (S \times T) \cup (T \times S)$ *are the arcs.*

We can work on Labeled Petri Nets (LPNs) adding an alphabet A and a labeling function $\lambda : T \to A$ to simple Petri nets. For a transition t in T, we define $\bullet t = \{s \in S \mid (s, t) \in F\}$ and $t\bullet = \{s \in S \mid (t, s) \in F\}$. A marking for a Petri net is a function $M : S \to \mathbb{N}$ assigning to each place a number of tokens. A marked Petri net is a couple (P, M) where P is the net and M a marking for this net. Petri nets may have an interleaving (retrieving their marking graph, an LTS) or a true concurrency (retrieving an LES) semantics. The usual marking graph semantics is given thanks to the transition firing rule: given a transition t

and a marking M, t is fireable iff $\forall s \in {}^\bullet t\ M(s) > 1$. If t is then fired, we get the marking M' where:

- for each s in S, $M''(s) = M(s) - 1$ if $s \in {}^\bullet t$ and $M''(s) = M(s)$ otherwise.
- for each s in S, $M'(s) = M''(s) + 1$ if $s \in t^\bullet$ and $M'(s) = M''(s)$ otherwise.

I refer to [Reisig, 1985] for more details on Petri nets (including weights associated to arcs) and their marking graph semantics, and to [Winskel, 1986] for their LES semantics.

An important operation on Petri nets, in relation to composite systems, is *fusion*. This can operate either on places or on transitions. The former typically represents processes (the nets) sharing some common resources (the fused places). The later typically represents processes (the nets) synchronizing on some interactions (the fused transitions). An important benefit of Petri net is therefore not only to support a true concurrency semantics, but also to enable to model easily both data/resource-based interaction and communication-based interaction between components. To quote Van der Aalst [van der Aalst, 2005], *"one of the big misconceptions about Petri nets versus process algebras is that process algebras are compositional while Petri nets are not. [...] Petri nets can be used in a compositional way."*. Petri nets indeed have their own notions of modularity [Kindler and Petrucci, 2009] and compositionnality [Reisig, 2009], interestingly, closely related to interfaces and fusion.

Graph Planning (GP). Given an initial state of a propositional world, a goal state, and a set of actions operating on this world, planning [Ghallab et al., 2004] addresses the retrieval of action sequences called plans enabling to reach the goal from the initial state.

Definition 1.5.7 (Planning)**.** *Given a finite set $L = \{p_1, \ldots, p_n\}$ of proposition symbols, a planning problem is a triple $P = ((S, A, \gamma), s_0, g)$, where: $S \subseteq 2^L$ is a set of states, A is a set of actions, an* action *a being a tuple $(pre, effects^-, effects^+) \subseteq 2^L \times 2^L \times 2^L$ where pre, $effects^-$, and $effects^+$ denote respectively the preconditions, the negative effects ($effects^- \subseteq pre$), and the positive effects of a, γ is a state transition function such that, for any state s where $pre(a) \subseteq s$, $\gamma(s, a) = (s \backslash effects^-(a)) \cup effects^+(a)$, and $s_0 \in S$ and $g \subseteq L$ are respectively the initial state and the goal.*

A plan π is a sequence of sets of actions $\pi_1; \pi_2; \ldots; \pi_n$, in which each π_i $(i = 1, \ldots, n)$ is a set of paralleled actions (denoted with $||$). π_1 is applicable to s_0. π_i is applicable to γ (s_{i-2}, π_{i-1}) when $i = 2, \ldots, n$. $g \subseteq \gamma(\ldots(\gamma(\gamma(s_0, \pi_1), \pi_2)\ldots\pi_n)$. Different algorithms have been proposed to solve planning problems and get plans from them [Ghallab et al., 2004]. A particularly interesting approach is based on graph planning [Blum and Furst, 1997] that builds the state space in polynomial time.

Definition 1.5.8 (Planning Graph)**.** *A planning graph G is a directed acyclic leveled graph in which levels alternate proposition layers P_i and action layers A_i. The initial proposition layer P_0 contains the initial propositions (s_0). An action a is put in layer A_i iff $pre(a) \subseteq P_{i-1}$ and then $effects^+(a) \subseteq P_i$.*

Exclusion relations [Blum and Furst, 1997] are used in planning graphs to support negative effects. Non-exclusive actions added into one layer are independent in the sense

that they could possibly be executed parallelly. A planning graph actually explores multiple search paths at the same time when expanding. The construction of the planning graph stops at a layer A_k iff the goal is reached ($g \subseteq A_k$) or in case of a fixpoint ($A_k = A_{k-1}$). In the former case there exists at least a solution, while in the later there is not. Plans can be obtained from a planning graph using backward search from the goal.

1.6 Summary of the Contributions

In this document I present my recent work in providing solutions to the development of composite systems. Work I did relative to *in-the-small development*, addressing for example the integration of formal data type descriptions in behavioral languages [A4][3] is not presented here. I made the choice to focus on *in-the-large development* since I believe that this is the more important part in the development of composite systems. Moreover, I will not present the work I did on in-the-large modeling and verification [A3]. Rather, I chose to address support for *new activities* that have emerged from the *conjunction of the composition and reuse principles*, and this both in top-down and bottom-up development processes (Fig. 1.20): composition and adaptation (Chap. 2) on the one hand, and composition testing, realizability checking, and composition repair (Chap. 3) on the other hand.

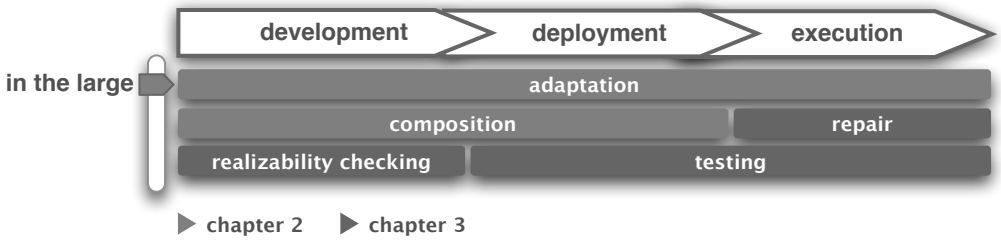

Figure 1.20: Composite systems activities presented in the manuscript

I will now give a summary of my contributions to these activities. These contributions will be detailed in Sections 2.1 and 3.1, together with a corresponding state of the art.

Semi-Automatic Adaptation. Reusable software entities are subject to mismatch preventing their correct composition. I have studied solutions to this issue through semi-automatic and automatic software adaptation approaches. I have first addressed **semi-automatic component adaptation**, Section 2.2. Based on adaptation contracts and component behavioral interfaces, a behavioral adaptor is constructed. The result of this work is a fully tool-supported centralized adaptation technique that combines the benefits of existing related work, being both restrictive and generative, and that apply system-wide, on any number of composed components. In order to enhance the efficiency of the adaptation process, incremental adaptation and a new kind of behavioral reduction have been studied too. Finally, in order to follow principles P5 and P6, application to a

[3]References of the form [Ax], [IPx] are personal ones, given in Chap. 5. Other ones, *e.g.*, [Allen and Garlan, 1997] are given in the bibliographic reference chapter.

real component model (Windows Workflow Framework) has been done. More recently, I have addressed **semi-automatic orchestration adaptation**, Section 2.3. This is an end-to-end (again, as stressed out by principles P5 and P6) application of adaptation to services with transformations from/to BPEL. Other contributions in this work include the direct support for data being exchanged and efficiency of the adaptation process through an on-the-fly technique for behavioral restriction and adaptor model reduction.

Automatic Adaptation. The work presented above is based on signature and behavioral interface descriptions only. Thus, it requires the existence of adaptation contracts in order to support solving out signature mismatch. In order to remove this constraint, I have studied **automatic distributed semantic adaptation**, Section 2.4. Using semantic annotations in service signatures, adaptation is performed at the information (data) level rather than at the message level. The bottleneck of centralized adaptation is avoided reusing local controller and projection techniques, yielding sets of local distributed adaptors that can be implemented as a set of distributed orchestrators, *i.e.*, as an adaptive choreography. More recently, this work has been extended using true concurrency models (LES) instead of interleaving ones (LTS). As a result, the transformation between global (centralized adaptor) and local (local adaptor) models is much simplified.

Composition. Adaptation and automatic composition are much related. Authors had shown that adaptation could go further than avoiding deadlocks and could be used to ensure system-wide properties. Still a remaining issue in composition is abstraction mismatch between composition requirements (activities or capabilities) and service description levels (operations). I therefore proposed techniques for **automatic composition of semantic services**, Section 2.5. With reference to existing work, the main contribution is to integrate adaptive features in the composition process, supporting conversations both for composition requirements and for service descriptions, and solving out both horizontal and vertical mismatch.

Testing. Abstraction used in the requirement, composition, and adaptation processes, together with the important role played by composition execution engines, make it important to test service compositions, whether they have been achieved manually, semi-automatically, or fully-automatically. In this direction, I have addressed **conformance testing of service orchestrations**, Section 3.2. The main contributions of this work are an end-to-end and fully automated online testing technique and a symbolic approach to avoid state space explosion in formal models due to the rich XML-based data used in Web service interfaces. Going further with composition in-the-large, **conformance testing of service choreographies** is addressed in Section 3.3. Most works address the testing of a privileged partner link in service orchestrations. Few works support the testing of choreographies. The first contribution is an end-to-end and fully automated testing technique for choreographies. With reference to related work it does not suppose access to the code since the approach is black-box based. Further, I propose to use passive testing in order not to be intrusive with the choreography under test.

Realizability checking. The two important issues with choreographies are conformance and realizability. In complement to my work on conformance testing presented above, I have proposed a technique for **choreography realizability checking**, Section 3.4. This

Table 1.2: Contributions - lecture grid

	issue	DL^{in}	DM^{in}	WM	DM^{out}	DL^{out}
adaptation and composition						
2.2	component adaptation	WWF	LTS	LTS	LTS	WWF
			LTS	PN	LTS	
2.3	orchestration adaptation	WSDL+BPEL	STS	PA	STS	BPEL
2.4	choreography adaptation	SAWSDL+BPEL	LTS	LTS	LTS	
		WF	LES	PN	LES	
2.5	orchestration composition	WF	GP	GP	GP	
verification and repair						
3.2	orchestration testing	ABPEL	STS	SET	STC	SOAP
3.3	choreography testing	Chor+SOAP logs	TSet	TSet	n/a	n/a
3.4	choreography checking	BPMN 2.0	WF	PA, LTS	n/a	n/a
3.5	orchestration repair	WSDL+OWL(+GP)	GP	GP	GP	BPEL

problem has been studied for interconnected interface models (MSC) and, more recently, for interaction interface models (UML collaboration diagrams). The main contribution is to study realizability on a more expressive choreography specification language, the new choreography diagrams introduced in BPMN 2.0.

Repair. Service compositions get broken as a consequence of users mobility, requirement change, and unavailable sub-services. As an alternative to solutions based on replacement or complete recomposition, I have proposed techniques for **service orchestration repair**, Section 3.5. Experiments on big size benchmarks constructed with a service composition challenge generating tool have demonstrated that repair gives solutions to broken compositions of equivalent quality but in better computation time.

These works propose a complementary set of solutions for the development of composite systems, as demonstrated in Figure 1.16 and Table 1.2. In the next two chapters, I will detail related work and my contributions (Sect. 2.1 and 3.1) together with the more important elements in the proposed solutions (Sect. 2.2–2.5 and 3.2–3.5). There are still interesting research directions to extend these solutions. This, together with other perspectives of my work is discussed in Chapter 4.

CHAPTER 2

Adaptation and Composition

2.1 State of the Art and Contributions

State of the Art and Contributions on Adaptation

The foundation for software adaptation has been set up in [Yellin and Strom, 1997]. Adaptation proposals initially focused on solving behavioral mismatches between abstract descriptions of *software components*. With the advent of *services*, these adaptation approaches have been extended and new ones have been developed to support this particular architecture. Several surveys have been published on software adaptation [Becker et al., 2006, Canal et al., 2006, Seguel et al., 2008, Inverardi et al., 2011]. One way to classify adaptation approaches is into *restrictive*, *generative*, and *ad-hoc* approaches [Canal et al., 2006].

Restrictive approaches, *e.g.*, [Reussner, 2003, Inverardi and Tivoli, 2003], solve mismatch out by cutting off the behavior that may lead to it, thus restricting the functionality of the involved components. These approaches can be related to controller synthesis. They are fully automatic and, often, tool-supported. However, they do not tackle advanced adaptation scenarios where message renaming is required and may have to change over time, or where reordering is needed. *Generative approaches*, *e.g.*, [Yellin and Strom, 1997, Bracciali et al., 2005, Brogi et al., 2006a, Brogi and Popescu, 2006], try to accommodate the protocols by generating adaptors that act as mediators, renaming, remembering and reordering messages and data when necessary. These approaches are able to solve more mismatch than restrictive approaches. However, for this they usually require an abstract specification of how mismatch can be solved, under the form of an adaptation contract, and are in this case only semi-automatic since a human intervention is required before the adaptor construction takes place. Generative approaches may also suffer from the computational complexity of generating adaptors due to reordering. A last cate-

gory of adaptation approaches are *ad-hoc approaches*, *e.g.*, [Schmidt and Reussner, 2002, Benatallah et al., 2005a, Dumas et al., 2006]. They address adaptation through solutions for specific mismatch with, *e.g.*, adaptation patterns or adaptation operator algebras.

An important issue in adaptation is to avoid the centralized adaptor bottleneck. A technique for the distribution of a centralized orchestrator into different topologies of decentralized orchestrators is presented in [Gowri Nanda et al., 2004] and extended in [Chafle et al., 2005] to support data flow constraints and a filtering mechanism to select the topologies that satisfy the constraints. These techniques still, do not address adaptation. In [Inverardi et al., 2005, Autili et al., 2008], the authors extend their earlier works on component adaptation to support the distribution of centralized adaptors introducing the concept of *last chance state* and using messages to ensure distribution correctness.

Over the years, adaptation approaches have greatly enhanced as far as expressiveness, mismatch support, automation, and relation to implementation languages is concerned. The SYNTHESIS framework [Inverardi and Tivoli, 2003, Autili et al., 2007, Autili et al., 2008, Tivoli and Inverardi, 2008] is a restrictive adaptation approach that goes beyond pure deadlock-freedom adaptation and takes into account additional adaptation inputs such as high-level MSC. In [Inverardi and Tivoli, 2003], it has been applied to COM/DCOM components, and application to services is tackled in [Inverardi and Tivoli, 2007] with orchestrator/sub-services correspondences that make it support a form of generative adaptation suitable to automatic composition. Another example is the [van der Aalst et al., 2009] approach that supports both restrictive and generative adaptation, ensuring weak termination (absence of deadlocks and absence of livelocks). Building on an important tool support based on Open Nets (a kind of Petri nets) and going beyond adaptation, this approach has been applied to services in [Gierds et al., 2010]. An adaptive composition technique based on the structural analysis of Petri nets (searching for siphons) is presented in [Xiong et al., 2010]. Mismatch can be avoided by adding new operations in the services when needed to control them, which yields an intrusive adaptation approach. Implementation in BPEL is also tackled in [Xiong et al., 2010]. Finally, a recent development in adaptation has addressed the construction of minimal adaptors [Seguel et al., 2010], *i.e.*, adaptors that are used only for a part of the component interactions.

The combination of behavioral and signature adaptation is receiving a growing interest, see, *e.g.*, [Motahari-Nezhad et al., 2007, Brogi and Popescu, 2007, Cavallaro et al., 2009, Motahari-Nezhad et al., 2010, Shan et al., 2010, Inverardi et al., 2011]. This is achieved mainly through the use of ontologies and semantic annotations in service signatures. These combined approaches often focus on specific usages such as service/client matching or service replacement, hence address binary adaptation. Few adaptation approaches address explicitly n-ary behavioral adaptation, *i.e.*, the construction of adaptors for any number of components or services. This is directly supported in the SYNTHESIS framework. In other purely behavioral adaptation approaches, *e.g.*, [van der Aalst et al., 2009, Gierds et al., 2010, Seguel et al., 2010], n-ary adaptation can be supported by adapting between a client and a system resulting from the product of all other components or services to adapt. As far as approaches combining behavioral and signature adaptation are concerned, techniques such as ontology integration could help in generalizing binary solutions for n-ary adaptation, as done, *e.g.*, in [Brogi et al., 2006b] for composition. Since the approach pre-

sented in [Inverardi et al., 2011] natively includes a form of behavioral ontology morphism, its extension to n-ary adaptation should be easier. Some authors have addressed adaptation with rich algebraic semantic descriptions [Morel and Alexander, 2004, Hemer, 2005]. The combination between these and behavioral adaptation is still to come.

A theoretical framework for adaptation with a process algebraic grounding is emerging [Padovani, 2009, Bravetti et al., 2011]. This can be related to behavioral types, contracts, or sessions [Vallecillo et al., 2003], a concise and expressive way to specify behavioral interfaces. They are of particular interest for discovery and selection as an alternative to LTS/graph-based techniques such as, *e.g.*, [Grigori et al., 2008]. Their use for adaptation, introduced in [Brogi et al., 2004], is promising for the incorporation of an adaptability notion in the discovery and selection processes, in complement to approaches based on compatibility measures [Ouederni et al., 2011].

Adaptive middleware is not a new topic [Agha, 2002]. It targets languages and infrastructures for adaptation, but not the generation of adaptor models and code. Still, one notes a convergence between software adaptation approaches getting more dynamic [Pelliccione et al., 2008, Cámara et al., 2008, Inverardi et al., 2011], with the related runtime issues, and recent development integrating (yet, simple) adaptation features in execution or monitoring tools [Wang et al., 2008, Moser et al., 2008, Taher et al., 2009]. The combination of expressive adaptation approaches with such support middleware is a promising direction for composition as a continuous process as presented in Chapter 1.

Contributions. In [IP16, A5] (Sect. 2.2) I have proposed techniques for the semi-automatic adaptation of software components that are both restrictive and generative. These techniques reused elements of two existing adaptation approaches, namely the removal of paths to deadlocks proposed in the [Inverardi and Tivoli, 2003] n-ary restrictive adaptation approach and the signature correspondences proposed in the [Bracciali et al., 2005] binary generative adaptation approach. I extended these binary correspondences into n-ary adaptation vectors to support n-ary generative adaptation. Further, using behavioral descriptions labeled with these vectors, it was possible to support complex adaptation policies, a simple form of adapted system properties, and, as opposed to adaptation where the contract is static, a first proposal for contextual adaptation contract. This last idea will be developed later on by other authors in [Cubo et al., 2007a]. I proposed two underlying algorithms. The first one is based on vector products. It is comparable to the adaptor generation approach proposed in [Inverardi and Tivoli, 2003], but for the support of name mismatch and evolving component signature morphisms using vectors. The first algorithm is more efficient than the second one, but it does not support generative adaptation due to its memory-less model. The second algorithm is based on an encoding into a formal model supporting message buffering, namely Petri nets. It is more complex but supports generative adaptation, including message storing and reordering. This second algorithm is acknowledged to be an inspiration for an extension of the SYNTHESIS framework to timed adaptation [Tivoli et al., 2007]. Following Principle P5 (see Sec. 1.4), the techniques I proposed were fully tool supported. Principle P6 called for experimentation on a real applicative domain. This has been done in [IP20, IP22] in relation to the Windows Workflow Foundation framework and with the semi-automatic implementation of adaptors in

.NET 3.0. My techniques were based on a centralized global adaptor for a closed system. I studied solutions to this with the incremental adaptation of open systems [IP17,IP18], still observing that in the general case open systems built incrementally require an overall adaptor to achieve deadlock freedom. Further, in order to reduce the size of adaptor models before implementation, I have proposed a new form of behavioral reduction for composite systems LTS models [IP19] based on the concept of equivalent transactions between stable (initial and final) states of a composition.

In a later work [IP21, IP25, A6] (Sect. 2.3) I have proposed an end-to-end adaptation approach for service compositions. It extended my previous work with the direct support for data in behavioral interfaces (using STS and value passing process algebras instead of LTS) and in adaptation contracts. Further, transformations from standard service description languages (WSDL, ABPEL) to STS service models, and from STS adaptor models to the standard service orchestration language (BPEL) have been defined (while omitted in most related work). Restrictive approaches proceed by removing paths leading to deadlocks (pruning). This can be a drawback when used in combination with generative approaches that generate a big state space due to the possible reordering of messages, even more if behavioral reduction is applied to adaptor models to reduce internal actions before implementation. To address this issue, a last contribution of this work is to perform pruning and reduction on-the-fly, *i.e.*, avoiding the complete construction of the global state space from the reused service models. This service adaptation approach is a core element of the ITACA adaptation and composition platform [Cámara et al., 2009].

All these works on adaptation required an adaptation contract to be given in order to support generative adaptation, hence were semi-automatic. In order to remove this constraint and achieve fully-automatic restrictive and generative adaptation, I studied in [IP23] (Sect. 2.4) the use of semantic annotations associated to data in the messages exchanged between services. In this work, adaptation is not performed at the operation level but rather at the information level, trying to exchange data between services in order to achieve partner satisfaction. As such, it is somehow a collaborative vision of adaptation, while the previous work was more concerned about cooperation. This work also yields distributed adaptors as opposed to a centralized one, following the steps introduced in [Inverardi et al., 2005] for stepwise distributed adaptor computation. Still, [IP23] builds on the extension of earlier work for service client synthesis [Haddad et al., 2004b], and then develops its own algorithms for client product and product projection. It should be noted that replacement algorithms for client synthesis [Inverardi and Tivoli, 2003] or for orchestrator/adaptor distribution [Gowri Nanda et al., 2004, Inverardi et al., 2005] could be reused through rephrasing in a semantic data centric framework. At the core of the [IP23] approach is the computation of a global adaptor model with implicit parallelism, hence resulting on an important interleaving and complexity of local adaptor retrieval and implementation. I recently studied in [S2] the use of a true concurrency model (LES) instead of LTS. As a result the transformation from global to local adaptors and the implementation of adaptors is much facilitated thanks to explicit concurrency.

State of the Art and Contributions on Automated Composition

Automated service composition fosters the rapid design, implementation and deployment of distributed applications. It is also a cornerstone towards the realization of on-demand composition from end-user requirements. Automated service composition has been studied under various assumptions as can be seen from recent surveys [Rao and Su, 2004, Dustdar and Schreiner, 2005, Marconi and Pistore, 2009].

Amongst the numerous techniques used for composition, *planning*, is increasingly applied in SOC [Peer, 2005, Chan et al., 2007] as it supports automatic service composition from underspecified requirements, *e.g.*, the data one requires and the data one agrees to give for this, or a set of capabilities one is searching for. Such requirements do not refer to service operations or to the order in which they should be called, which would be ill-suited to end-user composition. While both data input/output and capability requirements should be supported to ensure composition is correct wrt. the user needs, only [Bertoli et al., 2010, Ben Mokhtar et al., 2007] do. Most of the composition approaches indeed support only data [Klush et al., 2005, Constantinescu et al., 2006, Liu et al., 2007, Brogi and Popescu, 2005, Benigni et al., 2007, Zheng and Yan, 2008] or only capabilities [Berardi et al., 2004].

We have seen in Chapter 1 that composition requires adaptive features to solve horizontal and vertical mismatch. The former is supported in [Constantinescu et al., 2006, Liu et al., 2007, Ben Mokhtar et al., 2007, Benigni et al., 2007] using semantics associated to data. The later is supported in [Klush et al., 2005] due to its hierarchical planning inheritance. Few approaches support expressive models in which protocols can be described over capabilities, either for the composition requirement only [Berardi et al., 2004] or for both composition requirement and services [Bertoli et al., 2010, Ben Mokhtar et al., 2007]. [Klush et al., 2005, Constantinescu et al., 2006, Liu et al., 2007, Brogi and Popescu, 2005, Benigni et al., 2007] only support conversations over operations (for a given capability).

To be comprehensive, one should cite software adaptation works that are able to generate adaptors taking into account adaptation contracts *e.g.*, [A5] [van der Aalst et al., 2009], or high-level requirements and properties [Inverardi and Tivoli, 2003]. These works have been extended to services, in [A6], [Gierds et al., 2010], and [Inverardi and Tivoli, 2007], respectively. The main drawback of these works as far as composition is concerned is that they require that both requirements and service conversations are described at the operation level and rely on explicit operation correspondences. A fully automatic adaptive composition approach is proposed in [Xiong et al., 2010], but it requires to change the signature interface of the reused services, which is not suitable for black-box services.

Contributions. In my work [IP26, IP30] (Sect. 2.5), I use graph planning techniques to achieve composition from under-specified requirements. I propose an expressive framework with support for both data and capability based composition. While in [IP26] conversations were only supported in composition requirements, in [IP30] both requirements and service behavioral interfaces are given in terms of workflows. Using semantic annotations for data and capabilities, I introduce the necessary adaptation features in order to solve automatically both horizontal and vertical mismatch. Following my general principles (Sec. 1.4), the comprehensive encoding of the composition problem into a planning problem

enables to connect in [IP30] to efficient available planners rather than relying partly as in [IP26] on ad-hoc composition algorithms. To the difference of other composition approaches, including the ones based on adaptation or on planning, the use of graph planning brings benefits in case of broken compositions, *i.e.*, enabling repair (Sec. 3.5) instead of requiring full recomposition.

2.2 Centralized Adaptation of Software Components

[IP16,IP17,IP18,IP19,IP20,IP22,A5] – work done in the context of collaborations with colleagues from L'Aquila University, Italy, Málaga University, Spain, and Dauphine University.

Overview

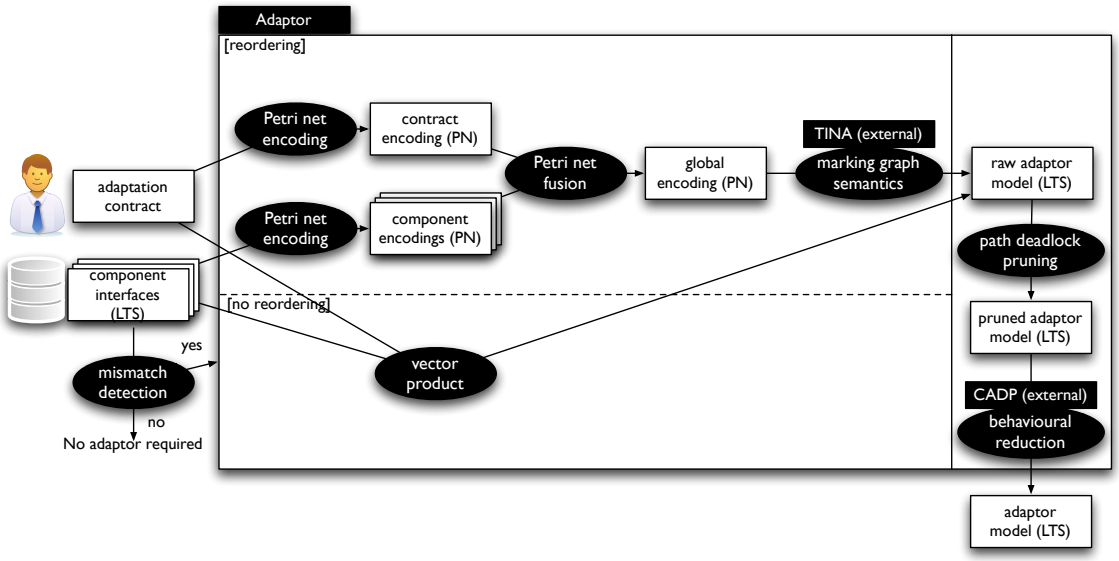

Figure 2.1: Centralized adaptation of software components – overview

This first adaptation approach (Fig. 2.1) takes as input the behavioral interfaces of components to be adapted and an adaptation contract, that is an abstract description of the constraints which must be respected to make the involved components work together. Given these two elements, an adaptor protocol is generated in an automatic way. The adaptation process begins with two (or more) components that are not able —as they are— to interact successfully (*i.e.*, ending in correct termination states). To compensate such mismatch, we propose to use synchronous vectors as adaptation contract language to make explicit the interactions between components, possibly on different message names. Our notation also allows the specification of ordering constraints on interactions,

which enables one to describe in an abstract way more complex adaptation scenarios. In order to generate adaptor protocols for such contracts, we propose two algorithms that automate the adaptation process. The first one is based on synchronous products, and the second one is based on Petri net encodings. Compared to the former, the latter induces a higher computational complexity, but is able to reorder messages when necessary, and then ensures a correct interaction when several components have the messages exchanged in their protocols which are not ordered correspondingly. Reordering is worked out desynchronizing the message emission by one component and the message reception in another one. When required, emitted messages are temporarily memorized until they are used for effective interaction. This is why a formalism capable of representing memory, such as Petri nets, is required. Our approach for software adaptation has been implemented in a tool, called Adaptor, and a Web service version of it, WS-Adaptor. Adaptor interacts with two other external tools, TINA (Petri net analysis) and CADP (mismatch computation and behavioral reductions).

Models

Component behavioral interfaces are represented by means of LTS where the alphabet corresponds to the input and output of messages (denoted with $m?$ and $m!$). For the adaptation contract notation, we rely on *synchronous vectors* [Arnold, 1994], which denote communication between several components, where each event appearing in one vector is executed by one component and the overall result corresponds to a synchronization between all the involved components. A vector may involve any number of components and does not require interactions to occur on the same names of events. Vectors can describe expressive communication patterns, which is especially useful to express n-ary interactions.

Definition 2.2.1 (Vector)**.** *A* synchronous vector *(or* vector *for short) for a set of components* $C_i = (A_i, S_i, I_i, F_i, T_i)$*,* $i \in \{1, \ldots, n\}$*, is a tuple* $\langle e_1, \ldots, e_n \rangle$ *with* $e_i \in A_i \cup \{\epsilon\}$*,* ϵ *meaning that a component does not participate in a synchronization.*

A vector can be implicitly given (no mandatory order as in a tuple, no ϵ) as an Id indexed set of alphabet elements $\{i : e_i\}, i \in Id$ with $e_i \in A_i$. The translation from one to the other is trivial. Vectors express correspondences between messages, like bindings between ports, or connectors, in architectural descriptions. Yet, vectors alone are not sufficient to perform adaptation as one must take into account also the context in which messages are exchanged, *i.e.*, the component protocols. Suppose we have a vector $\langle \mathsf{c_1} : \mathsf{a!}; \mathsf{c_2} : \mathsf{b?} \rangle$. Directly sending in an adaptor the message b to $\mathsf{c_2}$ when message a is received from $\mathsf{c_1}$ may lead the system to a deadlock state if this interaction is incorrect. This is why more complex adaptation algorithms, such as the ones we define here are required. Moreover, vectors are not sufficient to support more advanced adaptation scenarios such as contextual rules, choice between vectors or, more generally, ordering (*e.g.*, when one message in some component corresponds to several in another component, which requires to apply several vectors). The ordering in which vectors have to be applied can be specified using different notations such as regular expressions, LTSs, or (Hierarchical) Message Sequence Charts. Due to their readability and user-friendliness, we chose to specify adaptation contracts using *vector LTSs*, that is, LTSs

whose labels are vectors. In addition, vector LTSs facilitate the development of adaptation algorithms since they provide an explicit description of the contract behaviors set of states, which makes their traversal easier. Other notations, such as the ones mentioned above, can be used to specify the adaptation contract, provided that they can be translated into vector LTSs. To this purpose, one can rely on existing behavioral model synthesis techniques such as those presented in [Hopcroft and Ullman, 1979] for regular expressions, or in [Uchitel et al., 2003] for Message Sequence Charts.

Definition 2.2.2 (Vector LTS)**.** *A* vector LTS *for a set of vectors V is an LTS (V, S, I, F, T) where labels are vectors.*

Definition 2.2.3 (Adaptation Contract)**.** *An* adaptation contract *for a set of components $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, .., n\}$, is a couple (V, L) where V is a set of vectors for components C_i, and L is a vector LTS for V.*

There exists numerous definitions of compatibility and, as a consequence, of mismatch between protocols, but deadlock is the most commonly accepted notion. Still, the deadlock notion should take into account the specific role of final states. A system is blocked when it cannot evolve and when at least one of the components is not in one of its final states.

Definition 2.2.4 (Deadlock State)**.** *Let $C = (A, S, I, F, T)$ be an LTS. A state s in S is a deadlock state, denoted dead(s), iff it is not in F and it has no outgoing transitions.*

Definition 2.2.5 (Behavioural Mismatch)**.** *An LTS $C = (A, S, I, F, T)$ presents a behavioral mismatch if there is a state s in S such that dead(s).*

To check if a system composed of several components presents mismatch, its synchronous product with $\overline{m?} = m!$ is computed and then Definition 2.2.5 is used. An adaptor is given by an LTS which, put into a mismatching system yields a non-mismatching one. All the exchanged messages will pass through the adaptor, which can be seen as a coordinator for the components to be adapted. In the sequel, we present the principles of our two algorithms for the generation of adaptor protocols.

Synchronous Product Approach

We rely on an extension of the synchronous product that takes into account the correspondences of events described in the vectors, but also their ordering in the vector LTS. Consequently, the vector LTS is used as a guide to build the resulting product.

Definition 2.2.6 (Synchronous Vector Product (with vector LTS))**.** *The* synchronous vector product (with vector LTS) *of n LTS $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, .., n\}$ with a vector LTS $L = (A_L, S_L, I_L, F_L, T_L)$, is the LTS $\Pi_L((C_1, \ldots, C_n), L) = (A, S, I, F, T)$ such that:*

- $A = A_L \times A_1 \cup \{\epsilon\} \times \ldots \times A_n \cup \{\epsilon\}$, $S = S_L \times S_1 \times \ldots \times S_n$, $I = (I_L, I_1, \ldots, I_n)$, $F = F_L \times F_1 \times \ldots \times F_n$, *and*

- *T contains a transition $((s_L, s_1, \ldots, s_n), \langle a_L, a_1, \ldots, a_n \rangle, (s'_L, s'_1, \ldots, s'_n))$ iff there is a state $(s_L, s_1, \ldots, s_n)$ in S, there is a transition $(s_L, \langle l_1, \ldots, l_n \rangle, s'_L)$ in T_L and for every i in $\{1, \ldots, n\}$:*
  - *if $l_i = \epsilon$ then $s'_i = s_i$ and $a_i = \epsilon$,*
  - *otherwise there is a transition (s_i, a_i, s'_i) with $a_i = l_i$ in T_i.*

To generate an adaptor protocol from a synchronous vector product we have to discard the first element of the product components to keep only the elements corresponding to the component exchanges. More formally, it means that from an LTS $P_L = \Pi_L((C_1, \ldots, C_n), L) = (A, S, I, F, T)$ we compute the LTS $P = proj(P_L) = (A', S', I', F', T')$ such that $X' = \{cdr(x) \mid x \in X\}$ for any X in $\{A, S, I, F\}$, and $T' = \{(cdr(s), cdr(l), cdr(s')) \mid (s, l, s') \in T\}$ with $cdr((x_0, x_1, \ldots, x_n)) = (x_1, \ldots, x_n)$. To ensure deadlock freedom we then apply a technique for restrictive adaptation proposed in [Inverardi and Tivoli, 2003], pruning paths to deadlocks (Def 2.2.7).

Definition 2.2.7 (Path to deadlock)**.** *Given an LTS $C = (A, S, F, I, T)$, a path to deadlock is a sequence of transitions $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} s_n$ such that $deg(s_0) > 1 \wedge deg(s_n) = 0 \wedge \forall i \in \{1, \ldots, n-1\}\ deg(s_i) = 1$.*

Non controllable transitions in components can be supported in adaptor pruning using controller synthesis steps [Autili et al., 2008]. In our case, if the first transition in a path to deadlock is not controllable, we backtrack to remove controllable transitions related to the same component. We then mirror ($m? \leftrightarrow m!$) transitions to ensure that the adaptor and the components can perfectly communicate. Finally, we replace each transition labeled by a tuple of synchronized messages by all possible interleavings of these messages, starting by reception ones. This is essential when vectors involve more than two events in a communication (*e.g.,* in case of broadcast or multicast communication). Interleavings make the adaptor support non-determinism *wrt.* the orderings in which events will occur, hence accept any possible one.

Petri Net Approach

Our goal is now also to address behavioral mismatch which requires reordering. This occurs when exchanged messages present non-compatible orderings in the components' protocols. To support this kind of mismatch, the adaptation process may try to accommodate protocols by reordering events in-between the components. The behavioral adaptation proposal presented before may yield an empty adaptor in presence of such mismatch because it induces application of one vector after the other, and therefore prevents the application of several vectors at the same time that is necessary to make reordering effective. To this purpose, we present a second approach which complements the first one.

Messages received by the adaptor are seen as *resources* which are memorized until they need to be sent (*i.e.*, until they may be received by some component to make it evolve). This can be achieved first thanks to an *encoding* of the component protocols and of the adaptation contract into a formalism that supports a *memory* and a *resource-based vision of*

adaptation, as follows: (i) reception of messages (by the adaptor) corresponds to a resource creation, (ii) emission of messages (by the adaptor) is possible provided some resource is available and corresponds to resource consumption, and finally, (iii) vectors correspond to resource transfer. Petri nets are such a formalism, which further benefits from good tool support. Moreover, the marking graph of such a Petri net encoding represents all possible resource-based evolutions of the adaptor (message reception, emission and transfer).

Our second algorithm takes as input a set of component LTSs C_i and an adaptation contract, and generates first the corresponding Petri net encoding.

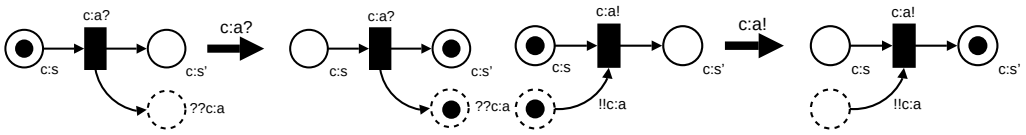

Figure 2.2: Encoding patterns for component protocols (and related semantics)

As regards component interface encoding (Fig. 2.2), every event emission or reception in a component is translated into a Petri net transition holding the same name as the event but the reversed direction. This transition is connected to specific places that are used to store, using tokens, messages corresponding to the events. For each event emission $c:a!$ in a component c interface there is a transition for reception in the Petri net ($c:a?$) and this transition has an output arc to the place where the corresponding message is stored ($??c:a$). Conversely, for each event reception $c:a?$ in a component c interface there is a transition for emission in the Petri net ($c:a!$) and this transition has an input arc from the place where the corresponding message has been stored ($!!c:a$). The control flow between events in component interfaces is expressed in the Petri net by control places ($c:s$ and $c:s'$) and related arcs connecting the different Petri net transitions. Moreover, tokens are placed in the control places encoding the initial states of the LTS interfaces, and their evolution will simulate the execution of the entire system.

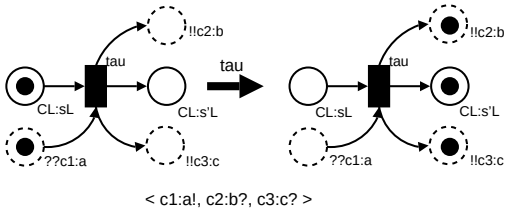

Figure 2.3: Encoding pattern for adaptation contracts (and related semantics)

As far as the contract encoding is concerned, every synchronous vector is encoded using a τ transition (Fig. 2.3) as it represents an internal action of the adaptor. Arcs are added to connect these τ transitions in order to enforce their application ordering in the vector LTS. Message transfer is enabled using input/output arcs that connect a τ transition to the places related to the component events involved in the corresponding vector.

To generate the adaptor protocol, we fuse all Petri net encodings (n for the components and one for the contract) on common places (the $??c : a$ and $!!c : a$ ones) and compute the marking graph of the fused Petri net. It contains all the possible evolutions of the adaptor *wrt.* the component LTSs it is in charge of. As for the previous approach, we remove paths to deadlocks. Finally, we apply a τ branching reduction to remove internal steps that are used in the encoding and adaptor generation and that are meaningless for adaptor implementation.

2.3 Centralized Adaptation of Web Services

[IP21,IP25,A6] – work done in the context of collaborations with colleagues from Málaga University, Spain, and INRIA Rhône-Alpes.

Overview

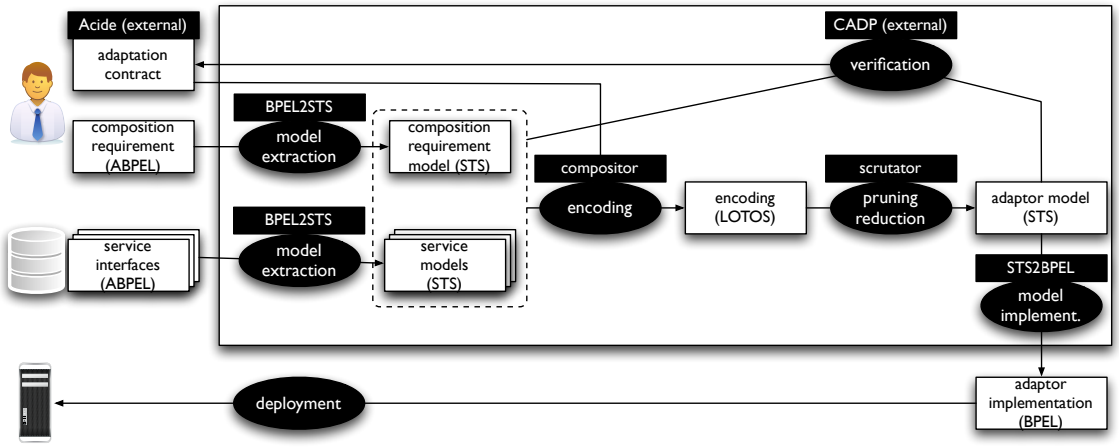

Figure 2.4: Centralized adaptation of Web services – overview

In this second adaptation approach, our goal was to support data in a more straightforward manner: they had to be encoded as additional messages in a preprocessing step with the previous approach. This augmented the size of the models being worked on. Further, we aimed at enhancing the efficiency of the adaptation process with on-the-fly deadlock pruning and adaptor reduction, as opposed to the previous approach requiring the construction of the complete state space. Finally, we wanted to achieve a comprehensive end-to-end approach for the service applicative domain with model retrieval from service descriptions, adaptor verification, and adaptor implementation in BPEL.

Our approach (Fig. 2.4) takes as input service behavioral descriptions and an adaptation contract. The descriptions are first transformed into STS formal models. Both these models and the contract are encoded into a value passing process algebra, namely LOTOS. This is

then used by our on-the-fly pruning and reduction tool to generate an adaptor STS model. This model can be verified and finally implemented by transformation into BPEL.

Our proposal is supported by tools that automate the generation of the encoding (Compositor), the efficient computation of the adaptor protocol from this encoding (Scrutator), and the model-based verification of the adapted system (here we reuse Evaluator and Bisimulator from the CADP toolbox). Relationship with BPEL is achieved with two tools that respectively extract abstract service models from XML descriptions of services (BPEL2STS) and support the generation of BPEL adaptors from adaptor models (STS2BPEL). The adaptation contract construction can be assisted and partially automated using recent results presented in [Martín and Pimentel, 2009, Cámara et al., 2009] and implemented in the Acide tool.

Models

A (Web) service signature is a combination of data structures (defined using XML Schema), operations, and messages (defined using WSDL). We model it as a tuple $\Sigma = (\mathcal{D}, \mathcal{O}, in, out)$ where $\mathcal{D}$ is a set of domains that correspond to (XML) types, $\mathcal{O}$ is a set of (provided) operations that are either one-way or two-way, and $in, out : \mathcal{O} \rightarrow \mathcal{D} \cup \{\bot\}$ denote respectively the input and output messages of an operation ($\bot$ when undefined, *e.g.*, $out(o) = \bot$ for any one-way operation o). In case of a composite service, several signatures are to be taken into account (one for each partner link). To support both simple and composite services in a unified way, we introduce partnerships. A partnership, ρ, is a set of signatures indexed by a set, PN, of partner names: $\rho = \{\Sigma_{i,i \in PN}\}$. For a simple service, ρ is a singleton that corresponds to the service signature.

Events are built accordingly to the signature model. An input event, $o?x$, with $o \in \mathcal{O}$ and x a variable whose domain is $in(o)$, corresponds to the reception of the input message of operation o. Similarly, we define output events $o!x$, the domain of x being $out(o)$. When messages have several parts, events are modified accordingly. Additionally, the τ event is used to denote non-observable internal computations or conditions. Events carry symbolic information (variables), but guards and actions are not useful in behavioral interfaces. This is why we rely on a simpler form of STS wrt. Chapter 1. A service is a couple $(\rho, \mathcal{B})$ made up of a partnership, ρ, and a behavioral description of its protocol defined with an STS whose alphabet corresponds to events built on the partnership signatures.

In order to specify adaptation requirements, we use again vectors and vector LTS. However, vectors are extended with support for data. Variables are used in the events of vectors as *placeholders* for message parameters. The same variable name appearing in different events (possibly in different vectors) enables one to relate sent and received message parameters, *e.g.*, $\langle \mathsf{c}_1 : \mathsf{a!sid}, \mathsf{x}, \mathsf{y}; \mathsf{c}_2 : \mathsf{b}_1\mathsf{?sid}, \mathsf{x} \rangle$ and $\langle \mathsf{c}_1 : \epsilon; \mathsf{c}_2 : \mathsf{b}_2\mathsf{?sid}, \mathsf{y} \rangle$. These correspondences are used in adaptor generation to solve data mismatches.

Process Algebra Encoding

Building an adaptor, one has to take into account different constraints. An adaptor has to act non-intrusively, *i.e.*, the interactions between an adaptor and a given service correspond

to this service's conversation. The adaptor has also to respect the system-level properties specified in the adaptation contract. In this work, we encode these adaptation constraints into a generic value-passing process algebra to foster genericity of the approach (see [A6] for details). In practice, we have used LOTOS [ISO/IEC, 1989]. Symbols ↑ (emission) and ↓ (reception) are supposed to be the symbols in the process algebra supporting data transfer. In our encoding, we have different kinds of processes:

- s_i:ServiceProtocol (one for each service): these processes encode the service conversations, *i.e.*, the ordering in which services accept or send messages (accordingly, the ordering in which the adaptor may send or receive these messages);
- :Store (one): this process encodes data availability, *i.e.*, which data have already been received or not by the adaptor, hence which can be used in output messages;
- :VectorLTS (one): this process encodes the adaptation contract, *i.e.*, it imposes the order in which vectors can be successively applied;
- v_k:Vector (one for each vector): these processes encode the vectors, *i.e.*, the input and output messages for the vector, together with the fact that input messages should be received before output ones.

Central to the way adaptation constraints are encoded as different processes is the way these constraints are related. Due to the interactive nature of these relations we present them using a sequence diagram (Fig. 2.5) where data are omitted for readability (however, they are taken into account in the encoding). Please note that with reference to the standard sequence diagram notation we introduce parallel blocks (par) to express that several interactions occur in parallel. This yields a more concise representation.

Processes synchronize to enforce ordering constraints over the reception/emission of messages. A specific action, FINAL, is used to denote correct termination of the processes. The fact that all processes end up synchronizing over FINAL denotes successful adaptation. We suppose for the sake of presentation that we are in the context of a vector LTS that prescribes the $v.v'$ sequence (v can be enabled and then v' will). Further, let v be a vector with n service outputs (*i.e.*, n adaptor inputs), $O = \{s_i : o_i\}$, $i \in \{1, \ldots, n\}$, and m service inputs (*i.e.*, m adaptor outputs), $I = \{s'_j : i_j\}$, $i \in \{1, \ldots, m\}$, where each o_i is of the form $opname_i!PH_{i,1}, \ldots, PH_{i,l_i}$ and each i_j is of the form $opname_j?PH_{j,1}, \ldots, PH_{j,l_j}$. For readability purposes, placeholders are not made explicit in Figure 2.5, where each arrow between a service process and a vector process (the o_i and i_j arrows) should be understood as a synchronization between the two processes. For example, the o_i arrow corresponds to a synchronization between $s_i : opname_i! \uparrow PH_{i,1}, \ldots, PH_{i,l_i}$ in the process for v and $s_i : opname_i! \downarrow x_1, \ldots, x_{l_i}$ in the process for s_i. Accordingly, the i_j arrow corresponds to a synchronization between $s'_j : opname_j? \uparrow PH_{j,1}, \ldots, PH_{j,l_j}$ in the process for v and $s'_j : opname_j? \downarrow x_1, \ldots, x_{l_j}$ in the process for s'_j. Our objective is to build the legal orderings in which such adaptor events can occur. For this we use the constraints of the adaptation problem as presented above. The main idea here is to use process algebra synchronized communication (represented with arrows in Fig. 2.5).

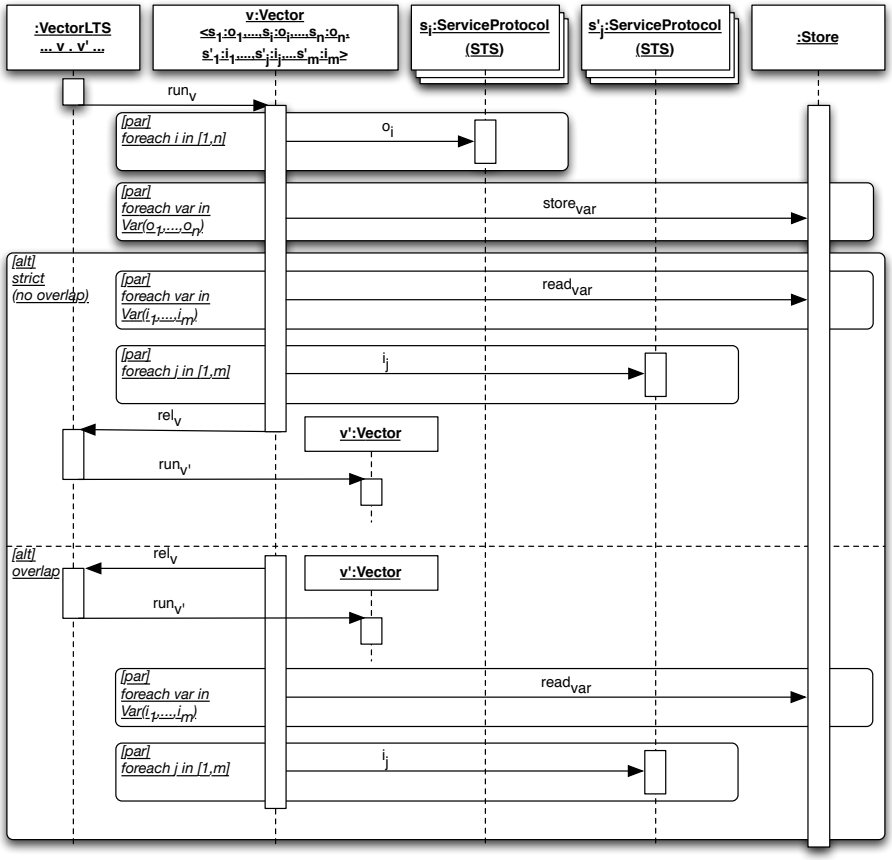

Figure 2.5: Encoding pattern

The vector v is first enabled through a run_v synchronizing with the vector LTS process, which gets suspended. Next, the vector tries to synchronize with all service outputs, O, since all the corresponding messages must have been received before the adaptor can send messages to the service inputs, I. Still, the messages for O may be received in different orderings. Since we want to allow any possible one, we define v to receive the messages for O in parallel (interleaving). Once all these messages are received, v sets the corresponding placeholders to true by communicating with the store process. As far as the messages for service inputs, I, are concerned, v checks first that all required placeholders have been set up (either by v or another vector). If not, v waits until another vector receives the service message(s) with the pending placeholders. Once all placeholders are available, v may execute the I actions, again in any possible order, by synchronizing with the corresponding services.

When a vector has been completely executed, it can be run again once enabled by the vector LTS. However, several strategies are possible for a vector v process to release a suspended vector (using rel_v). The first strategy is to wait for the complete processing of a vector before launching a new one (Fig. 2.5, *strict* alternative block). The second one is to

execute the release action once the reception of service messages has been done and before doing the sending part (Fig. 2.5, *overlap* alternative block). The latter is particularly interesting since it makes the reordering of messages possible, a typical case of mismatch between services. However, there is no silver bullet: overlap yields larger state spaces when computing the adaptor models. Once an adaptor has been computed from this encoding (see below), communicating actions are relabeled removing ↑ and ↓ symbols since they are encoding-related only. Further, communicating actions are mirrored (*i.e.* ? and ! are reversed) since an adaptor mirrors service events.

Adaptor Generation

An adaptor can be obtained from the encoding by pruning paths to deadlocks in the explicit, *entirely constructed*, state space. To increase efficiency, we avoid the entire construction of the state space and instead we explore it forwards in order to generate the adaptor *on-the-fly* by carrying out deadlock elimination and behavioral reduction simultaneously.

First, the execution sequences leading to deadlocks must be pruned. We do this by keeping, for each state encountered, only its successor states that potentially reach a successful termination state, which is source of a transition labeled with the action FINAL. Besides avoiding deadlocks (sink states reached by actions other than FINAL), this also avoids livelocks, *i.e.*, portions of the state space where some services get "trapped" and cannot reach their final states anymore. The desired successor states satisfy the μ-calculus [Kozen, 1983] formula $\mu X. \langle\ \mathsf{FINAL}\ \rangle\ true \vee \langle\ true\ \rangle\ X$. An efficient solution is obtained by translating the evaluation of this formula into the resolution of an equivalent boolean equation system (BES) [Andersen, 1994]: $\left\{ X_s =_\mu \bigvee_{s \xrightarrow{\mathsf{FINAL}} s'} true \vee \bigvee_{s \to s''} X_{s''} \right\}_{s \in S}$. This BES consists of a set of minimal fixed point equations, each one defining in its left-hand side a boolean variable X_s indexed by a state $s \in S$ and having in its right-hand side a disjunctive boolean formula expressing whether s satisfies the modal formula or not.

The use of BES to encode deadlock elimination allows to perform this task on-the-fly, using the Caesar.Solve library [Mateescu, 2006] of CADP, during a forward exploration of the state space, and to store only states in memory. This is nearly optimal from the on-the-fly perspective, since in the worst case (*i.e.*, when the system contains no deadlocks) one has to explore the whole state space in order to generate the adaptor.

In the same way, the adaptor STS obtained by pruning can be reduced on-the-fly (simultaneously with its generation) modulo an appropriate equivalence relation in order to get rid of the internal τ actions resulting from the encoding of the adaptation constraints (*e.g.*, *run* and *rel*) and obtain an adaptor as small as possible.

Experiments have shown that the state space actually explored on-the-fly during the generation of the reduced adaptor can be significantly smaller than the whole state space (down to 51% of the states and 18% of the transitions). For examples that have small or medium sizes, these gains have a limited impact; however, they can lead to significant memory savings for larger systems. Moreover, we observe that the ratio concerning the number of transitions explored is smaller than the ratio concerning the states. This aspect — which becomes more important with the increase of the branching factor of the state space, proportional to the number of services in the adapted system — further penalizes

explicit approaches wrt. the on-the-fly approach, because the former requires to store all transitions in memory in order to compute the adaptor, whereas the latter allows to store only states.

Adaptor Verification

Adaptors respect their adaptation contracts and are free of deadlocks. However adaptation contracts are an input of the adaptation process, and their writing requires human intervention. Since contracts are specified by the designer, they can contain errors that will also appear at the level of the adaptor. Indeed, this specification may not correspond exactly to what the designer expects from the system-to-be. Therefore, we have proposed verification techniques for the adaptation contracts (placeholder and event consistency) and for adaptor models (placeholder and message occurrence, safety and liveness properties, adaptability checking in relation with non controllable service actions).

Relating Languages and Models

We have applied our adaptation technique to Web services through transformations between service description (WSDL and BPEL) and STS. For the transformation from WSDL and BPEL to STS, we take inspiration from our previous work on WWF [Cubo et al., 2007b]. Our transformation rules (detailed in [A6]) are also closely related to the recent work in [Marconi and Pistore, 2009].

Due to the model-driven approach, an adaptor model may contain parts which are not implementable in a given language (here, BPEL). Take the example in Figure 2.6 with two services (top and bottom left) and a user need (bottom right).

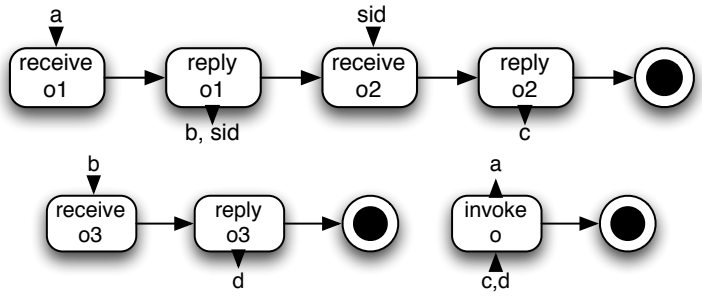

Figure 2.6: The need for filtering – motivating example (BPMN-inspired graphical notation for BPEL)

A corresponding adaptor is given in Figure 2.7. First, any path passing through state x cannot be implemented, since o_2 (resp. o_3) being two-way, one cannot invoke o_3 (resp. o_2) before getting o_2 (resp. o_3) result. Further, the condition to choose between invoking first o_2 and then o_3 or the converse is not specified. Indeed they could be invoked in parallel which is not explicit on an LTS/STS model. This is to be further discussed in the next section.

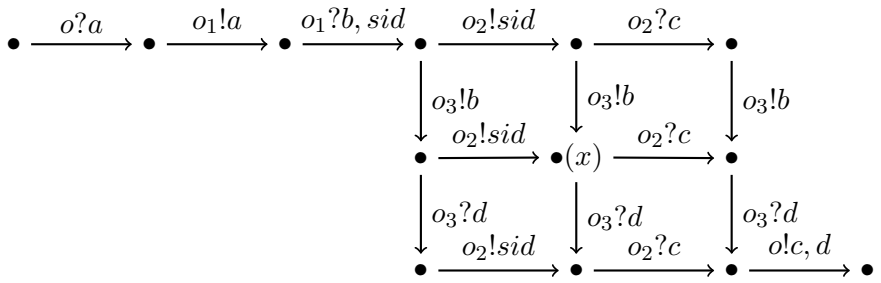

Figure 2.7: The need for filtering – adaptor LTS model

Therefore, adaptor implementation is achieved in three steps: filtering, model cleaning, and generation. Filtering ensures emission determinism, priority of receptions over emissions, and atomicity of the emission and reception corresponding to the invocation of a two-way operation. Model cleaning is then used to ensure that deadlocks are not introduced by filtering. Filtering may remove parts of the adaptor model. If deadlock-freeness is preserved, some properties of the adaptors may change. It is therefore interesting to verify the filtered adaptors, which can be done using our verification techniques. Finally, filtered and cleaned adaptor models are encoded in BPEL using a specific design pattern, namely the state machine pattern.

2.4 Distributed Adaptation of Semantic Web Services

[IP23,S2] – work done in the context of collaborations with colleagues from Evry University and INRIA Paris-Rocquencourt.

Overview

In this work our first objective is to enable generative adaptation without requiring the need for adaptation contracts, and as a consequence enable fully-automatic restrictive and generative adaptation. Further, we take a data centric perspective (as opposed to a message centric one) over adaptation. The issue is to have partner services exchange information in such a way that each one may progress in its objectives (*i.e.*, collaboration), rather than performing mitigation at the message level to achieve a global goal (*i.e.*, cooperation). For this, we rely on semantic annotations. This corresponds to a form of additional typing and a transfer from an in-the-large software architect work (defining adaptation contracts) to an in-the-small service developer work (adding annotations in service interfaces). Our second objective was to avoid centralized adaptors and enable distributed adaptation.

Given n semantic services to compose (the partners), our approach (Fig. 2.8) follows the stepwise approach for adaptor synthesis proposed in [Inverardi and Tivoli, 2003], rephrasing it in a semantic data centric perspective, with three distinct steps: (i) the generation of a compatible (correct) service client (CSC) for each partner service, which allows to interact

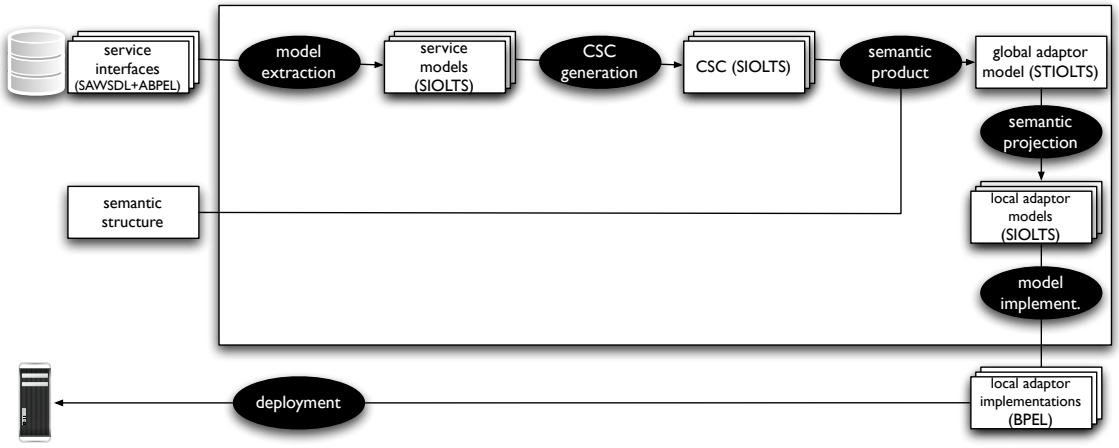

Figure 2.8: Distributed adaptation of semantic Web services – overview

with the partner without changing its protocol, (ii) using the CSCs, the definition of a central global adaptor that defines all the valid interactions between partners, and finally (iii) the transformation of CSCs into local adaptors using the global adaptor protocol. The final desired architecture follows the distributed adaptation pattern (Fig. 1.12, right) and is such that there is compatibility between each partner and its local adaptor and between the adaptors. This yields the deadlock freedom of the global system.

Models

To support automatic composition, service descriptions must be extended with descriptive semantic information. The resulting integrated formal model allows reasoning on service compatibility at three levels at the same time (signature, behavioral and semantic levels).

Definition 2.4.1 (Semantic Structure)**.** *A* semantic structure $\mathcal{I}$ *is a couple* $(\mathcal{U}, \mathcal{R})$ *where* $\mathcal{U}$ *is a set of units of sense (*UoS*), over which we range using* u*, and* $\mathcal{R} \subseteq 2^{\mathcal{U}} \times \mathcal{U}$ *is a relation where* $(U, u) \in \mathcal{R}$ *denotes that given a set* U *of UoS, one can obtain* u*.*

These structures support partner collaboration and can be related to concrete ontologies referenced, *e.g.*, in a SAWSDL [W3C, 2007] service description, where units of sense correspond to the ontology concepts and properties, while $\mathcal{R}$ can be used to encapsulate relations such as the "subclassOf" one, *i.e.*, $\forall u, u' \in \mathcal{U}$, u' subclassOf $u \Rightarrow (\{u'\}, u) \in \mathcal{R}$. A semantic structure may result from ontology integration, *e.g.*, following [Brogi et al., 2006b], and support different semantic structures for different partners.

A service receives requests, process them and sends answers back. Yet, to process a request, behind its representation (the message format), a set of information, UoS, is required. In turn, replies contain (possibly new) UoS. To support the automatic use of such a service by a partner, one has to ensure that all required information is known by the partner, format the request message and package the information into it, call the service, get the response, and finally process it to extract the set of information that is returned

back. This principle is at the core of our adaptor behaviors and is supported through semantic matching functions.

Definition 2.4.2 ((Semantic) Matching Function)**.** *A* matching function *for a service ω with messages M_ω over a semantic structure $\mathcal{I} = (\mathcal{U}, \mathcal{R})$ is a function $SM_{\omega,\mathcal{I}} : M_\omega \rightarrow 2^{\mathcal{U} \times Xpath(M_\omega)}$ with $Xpath(M_\omega)$ the set of Xpath expressions defined over M_ω.*

These functions are used to associate to each message in a service ω the set of UoS it corresponds to, together with a syntactic expression ($Xpath$) that makes it possible to relate these UoS with the message XML tree. We also introduce a formal definition of *partners* and *partnerships* as a set of services collaborating on top of a semantic structure.

To support automatic adaptation and composition, the operational semantics of a partner should also be defined through its evolution over time, directed by its behavior, of hypotheses on the UoS it holds. This can be described using *configurations*.

Definition 2.4.3 (Configuration)**.** *Given the set ABP of abstract BPEL processes and a semantic structure $(\mathcal{U}, \mathcal{R})$, a configuration is a tuple $(P, \mathcal{H})$ where $P \in ABP$ is the process and $\mathcal{H} \in 2^{\mathcal{U}}$ is its semantic environment.*

The operational semantics of partners is then defined using Semantic TIOLTS (STIOLTS). This is a kind of LTS where states are configurations and events include termination, discrete time passing, non observable events, and disjoint input and outputs events for the partner messages. We define an ABPEL to STIOLTS transformation that extends an ABPEL to TIOLTS transformation [Haddad et al., 2004b] with semantic data evolution constraints: given a service ω and a message m, the reception of m in ω augments its set of known UoS ($\mathcal{H} \leftarrow \mathcal{H} \cup SM_{\omega,\mathcal{I}}(m)$) and the emission of m by ω is constrained by the fact that the UoS for m should be known ($SM_{\omega,\mathcal{I}}(m) \subseteq \mathcal{H}$).

Generation of Correct Service Clients

We compute, for each partner ρ, a CSC ρ^c which acts as a perfect environment for it and ensures compatibility by construction. Two tasks are performed on a partner STIOLTS to build its CSC STIOLTS: messages are complemented (exchanging directions) and the resulting STIOLTS is determinized, following [Haddad et al., 2004b]. At the time, a CSC does not really provide the required operations to its partner, but only the corresponding received and/or send messages. The CSC must now be *extended* with implementations of these operations in terms of interactions with other CSCs. This is the objective of the next steps, where a global structure (called global adaptor) is first built (step ii) before being used to extend the CSCs into local adaptors (step iii).

Generation of a Global Adaptor

Composing partners correctly in a partnership means to have their CSC exchange required data until all terminate. The global adaptor (GA) is a composition of the CSC STIOLTS formalized as an extension of the asynchronous product of LTS (Chap. 1) supporting the data evolution constraints presented above. Moreover, in this extended asynchronous

product termination events, $\surd$, are synchronized to ensure correct termination. The GA has access to UoS originating from (different messages of) different partners. We need to distinguish them in order to support step (iii): when adaptation is distributed, we must be able to detect which messages were used to obtain a UoS. Therefore, in the GA configurations, the semantic information is stored as couples (m, u) where u is an UoS and m is the message (that uniquely denotes a partner) the information comes from. In order to achieve restrictive adaptation, pruning paths to deadlock on the GA is performed.

Generation of Local Adaptors

The GA defines all valid interactions. Using it, we can generate for each partner ρ a *local adaptor* (hereafter adaptor for short) by extending its CSC ρ^c in three steps: (a) defining, for each message m sent by ρ^c to ρ, the set of UoS from other CSC used to construct m, and receiving in ρ^c these UoS (using one or several new messages) before sending m; (b) defining, for each message m received from ρ in ρ^c, the set of interested CSC that need UoS from m to send some message to their own partner, and sending in ρ^c the UoS to these interested CSC (using a new message also named m); (c) updating the ρ^c alphabet and operations. All these operations can be done by analyzing GA.

Step (a) is performed first using backward analysis in GA, from the sending of m by ρ^c to ρ, back to the messages that made the UoS (u) required for building m available. Yet, it is possible to look only one step back thanks to the way we build GA configurations (couples (m', u), where m' is the message u comes from, sent to GA by some ρ'^c). If a UoS may come from several messages (and/or several CSC) then we non-deterministically choose one. We may then obtain the extended behavior of ρ^c by adding a new event for the reception of m' from ρ'^c in the alphabet of ρ^c and then by projecting GA on this alphabet: this will include the original behavior of ρ^c plus the reception of m' at any possible moment where ρ'^c may send it. Hence no deadlock may be introduced by these new exchanges. Correspondingly, in step (b), an adaptor ρ^c must forward any message m received from its partner ρ to the adaptors that need it. This is achieved using GA forward analysis and the results of step (a). The last step, (c), is to add provided/required operations in partners accordingly to the extensions of their behavioral model alphabets done in steps (a) and (b). The adding of receptions and emissions in CSCs is transparent for their partners. Moreover, added emissions do not change the emitter CSC configuration. This preserves in adaptors the CSC correctness by construction. Our approach does not impose any constraint on resulting adaptors but for communication, data extraction from messages, and message construction (the latter two being supported by the Xpath part in matching functions), which makes it possible to reuse implementation techniques presented before.

Extension to True Concurrency Models

In a recent work [S2], this work has been extended to the use of a true concurrency model, namely LES, instead of (STIO)LTS as the formal grounding. The main differences are the following. First, LES are retrieved from an abstract workflow language rather than ABPEL. The retrieval of a GA using an LTS product is no longer needed. This is naturally

supported in LES using the union operator, the GA being replaced by the union of the CSC LES. Further, parallelism was implicit in the former approach. This was especially an issue in GA (yielding an important number of interleavings) and in local adaptors (where interleavings where relative to the added receptions of required information, step (a) above).

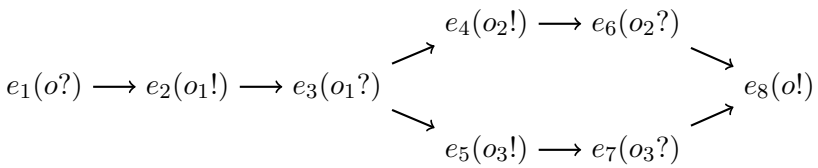

Figure 2.9: No need for filtering - adaptor LES model

Using LES, parallelism is explicit in the whole adaptation process. This is of particular interest for adaptor implementation as can be seen in Figure 2.9 that is the LES that corresponds to the adaptor LTS given in Figure 2.7. Here, it is easy to detect that the invocation of o_2 (two messages) and the invocation of o_3 (two messages) can be done in parallel, and *e.g.*, implemented as a BPEL flow.

2.5 Automatic Composition of Semantic Web Services

[IP24,IP30] – work done in the context of collaborations with colleagues from Concordia University, Canada, and INRIA Paris-Rocquencourt.

Overview

The objective of this composition approach is to support fully-automatic composition, expressiveness in service descriptions and composition requirements, and adaptive features to solve both horizontal and vertical mismatch. This is performed (Fig. 2.10) first by generating and encoding into planning problems the composition process inputs: semantic descriptions, service descriptions including a conversation given as an operation workflow, and composition requirements including a conversation given as an abstract capability workflow. These planning problems are then joint into a global planning problem that is solved using a graph planner. Finally, the output of the planner can be transformed into a BPEL orchestration. Our composition approach is supported with a tool, pycompose, that supports the use of several state-of-the art graph planners, including one based on SAT-solving for plan extraction.

Models

As in the previous work, we also rely on semantic information to enrich service descriptions and composition requirements, and to automatize the composition process. We have two

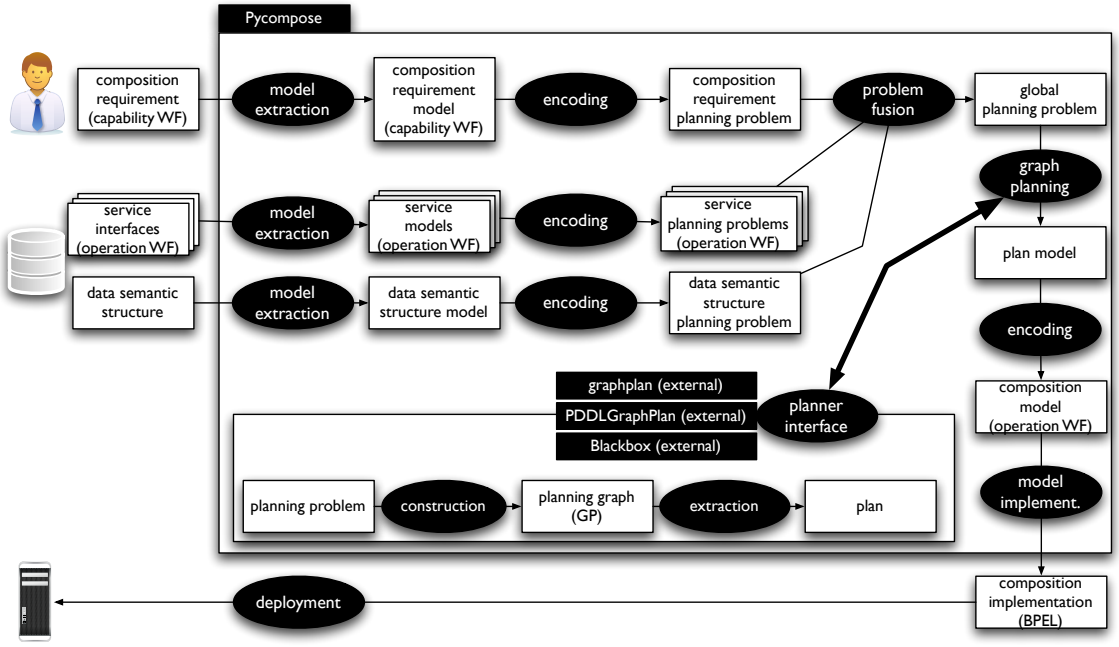

Figure 2.10: Automatic composition of semantic Web services – overview

kinds of semantic information. Capabilities represent the functionalities that are either requested by the end-users or provided by services. They are modeled using a Capability Semantic Structure (CSS), that support the resolution of vertical mismatch. Further, service inputs and outputs are annotated using a Data Semantic Structure (DSS), that support the resolution of horizontal mismatch.

Definition 2.5.1 (Data Semantic Structure)**.** *A* Data Semantic Structure (DSS) *is a tuple* $(\mathcal{D}, \lhd, \sqsubset)$ *where* $\mathcal{D}$ *is a set of concepts that represent the semantics of some data,* $\lhd$ *is a composition relation* ($(d_1, x, d_2) \in \lhd$*, also noted* $d_1 \lhd_x d_2$ *or simply* $d_1 \lhd d_2$ *when* x *in not relevant for the context, means a* d_1 *is composed of an* x *of type* d_2*), and* $\sqsubset$ *is a subtyping relation* ($d_1 \sqsubset d_2$ *means* d_1 *can be used as a* d_2*).*

DSSs are the support for the automatic decomposition (of d into D if $D = \{d_i \mid d \lhd d_i\}$), composition (of D into d if $D = \{d_i \mid d \lhd d_i\}$) and casting (of d_1 into d_2 if $d_1 \sqsubset d_2$) of data types exchanged between services and orchestrator. We also define a *Capability Semantic Structure (CSS)* as a set $\mathcal{K}$ of concepts that correspond to capabilities. This set includes an element $\perp$ that denotes a do-nothing capability.

A service is a set of operations described in terms of capabilities, inputs, and outputs. Additionally, services have a conversation defined using an operation workflow (Chap. 1).

Definition 2.5.2. *Given a CSS* $\mathcal{K}$ *and a DSS* $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$*, a* service *is a tuple* $w = (O, WF^O)$*, where* O *is a set of operations, an operation being a tuple* (in, out, k) *with* $in \subseteq \mathcal{D}$*,* $out \subseteq \mathcal{D}$*,* $k \in \mathcal{K}$*, and* WF^O *is a workflow defined over* O*.*

Finally, service composition requirements are given in terms of the inputs the user is ready to provide and the outputs this user is expecting. Additionally, the capabilities that are expected from the composition are specified, and their expected ordering given under the form of a capability workflow.

Definition 2.5.3. *Given a CSS $\mathcal{K}$ and a DSS $\mathcal{D} = (\mathcal{D}, \lhd, \sqsubset)$, a* composition requirement *is a tuple $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ where $D_{\text{in}} \subseteq \mathcal{D}$, $D_{\text{out}} \subseteq \mathcal{D}$, and $WF^{\mathcal{K}}$ is a workflow defined over $\mathcal{K}$.*

Planning Problem Encoding

We give the details of the encoding in [IP30]. Here we present only the general ideas of it.

DSS encoding. For each relation between data in the DSS, we generate one or two action(s) that represent the effect of this relation. For each $d \lhd \{x_i : d_i\}$ in the DSS we have an action $comp_d(\bigcup_i \{d_i\}, \emptyset, \{d\})$ and an action $dec_d(\{d\}, \emptyset, \bigcup_i \{d_i\})$ to model possible (de)composition. Moreover, for each $d \sqsubset d'$ in the DSS we have an action $cast_{d,d'}(\{d\}, \emptyset, \{d'\})$ to model possible casting from d to d'. These actions do not have negative effects since once a data is known or has been computed, it is never destroyed.

Workflow encoding. We reuse here a transformation from workflows to Petri net defined in [Kiepuszewski, 2003]. Instead of mapping a workflow to a Petri net, we map it to a planning problem. Given a workflow $W^A = (P, \rightarrow, N)$, we have an action for each node in P and for each control flow element in $\rightarrow$. Each of these actions requires, to be enabled, that some proposition(s), B^-, are available. Then, the action consumes these propositions and creates new ones, B^+, that will enable subsequent actions in the workflow. This corresponds to actions being encoded as (B^-, B^-, B^+).

Composition requirements encoding. A composition requirement $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$ is encoded as follows. D_{in} and D_{out} are used to set up the initial state and the goal of the planning problem. Then, we compute the set of actions resulting from the encoding of $WF^{\mathcal{K}}$. We also have to deal with two constraints (Fig. 2.11). First, capabilities in the composition requirement workflow ($WF^{\mathcal{K}}$) encoding prescribe when operations in service workflow ($WF^{O}(w)$) encodings can be done. In turn, these operations, when done, let further capacities to be enabled. To achieve this, we rely on two requirement/service interaction propositions: e_k (k enabled) and d_k (k done). A third kind of propositions, $link_a$, is used to avoid incorrect interactions in case of parallel flows [IP30]. We replace any action a (corresponding to a workflow activity labeled by k) in the encoding of $WF^{\mathcal{K}}$ by two actions, a' and $\overline{a'}$, and we set $a' = (pre(a), effects^-(a), \{e_k, link_a\})$ and $\overline{a'} = (\{link_a, d_k\}, \{link_a, d_k\}, effect^+(a))$.

Service encoding. Each service $w = (O, WF^O)$ is encoded as follows. First we encode the workflow WF^O. Then, for each action a (corresponding to a workflow activity labeled by o) in this encoding we add (Fig. 2.11):

- $in(o)$ in $pre(a)$ to model the inputs required by operation o and $out(o)$ in $effect^+(a)$ to model the outputs provided by operation o.

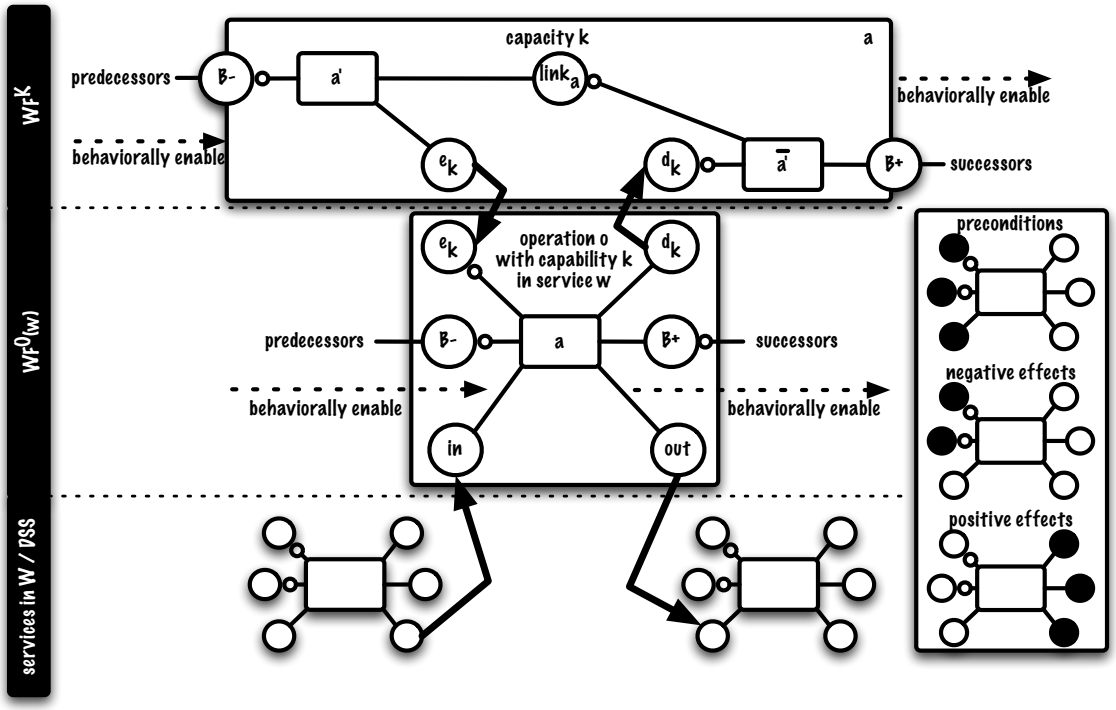

Figure 2.11: Principle of the interaction between requirements and service actions

- $e_{k(o)}$ in $pre(a)$ and in $effect^{-}(a)$ and $d_{k(o)}$ in $effect^{+}(a)$ to implement the interaction with the composition requirement workflow.

Overall encoding. Given a DSS $\mathcal{D}$, a set of services W, and a composition requirement $(D_{\text{in}}, D_{\text{out}}, WF^{\mathcal{K}})$, we obtain a planning problem doing the union of all encodings (some renaming being performed to avoid name clashes), by setting the initial state of the planning problem to be D_{in} plus all propositions encoding workflows initial states, and by setting the goal of the planning problem to be D_{in} plus all propositions encoding the workflows final states.

Planning *vs.* Petri nets. Our approach could be rephrased replacing the encoding into actions by (i) an encoding into Petri nets, (ii) the fusion of these Petri nets on common places, and (iii) the encoding of any transition t in the fused Petri net by a planning action $a(t) = (\bullet t, \bullet t \backslash t\bullet, t\bullet)$. The benefit of using planning graphs instead of Petri nets is the use of efficient planners, *e.g.*, based on SAT solving, for plan retrieval and the possibility to reuse partially broken planning graphs for composition repair (Sec. 3.5).

Plan Retrieval and Implementation

We solve the composition problem by applying graph planning (Chap. 1) on the encoding. We may get a failure when there is no solution satisfying both that (i) a service composition

exists to get D_{out} from D_{in}, (ii) using operations/capabilities in an ordering satisfying both used service operation-based conversations and composition requirements capability-based conversation, (iii) leaving used services in a final state. In other cases, we obtain a plan π. First, we filter π by removing from it all actions that correspond neither to a DSS relation nor to a service operation, *i.e.*, that is a purely structuring item, not corresponding to data transformation or service invocation. Given the filtered plan, we can generate a BPEL implementation for it as done for transitions systems in the previous sections. Still, we may benefit here from the fact that actions that can be done in parallel are explicited in a graph planning plan (using operation $||$).

CHAPTER 3

Verification and Repair

3.1 State of the Art and Contributions

State of the Art and Contributions on Testing

The state of practice in service testing has been limited for a long time to the use of tools, such as SOAPUI or BPELUnit, that release from the burden of the translation of implementation independent test cases into SOAP messages, operation calling and test management. However, test-cases were mainly generated using empirical approaches, often without automation. In the last years, the software testing community has started to get involved in the service field. As a consequence, works have tried to bridge the gap between current practice in service testing and state-of-the-art formal and automated software testing. Recent and comprehensive surveys can be found in [Bucchiarone et al., 2007, Canfora and Penta, 2009, Bozkurt et al., 2010, Endo and Simao, 2010, Rusli et al., 2011][1].

A first way to test services is to focus on signature interfaces (WSDL). These approaches enable to test operations independently but do not tackle testing of services with conversations or orchestrations. Taking into account behavioral descriptions can be studied using source code coverage criteria. However, such white-box approaches assume that the implementation source code is available. This is suitable for testing at development time by service providers but not later on for service users that only have access to the binary implementation. For the numerous proposals that fall in these two categories, we refer to the above mentioned surveys. In our context, we are interested in black box testing related to the behavioral conformance of an implementation with reference to a specification.

An important issue in Web service testing is the treatment of the rich structured data being used in service exchanges. This may be abstracted away but at the risk of over-approximation and testing false negatives. Such data cause state explosion

[1] Comparison tables are given in [Bozkurt et al., 2010, Rusli et al., 2011] and include reference to [IP26].

problems. Few approaches for behavioral testing of services propose solutions to this. In [Sinha and Paradkar, 2006], extended finite state machines are retrieved from WSDL-S, a semantic extension of WSDL including pre/post conditions, and a theorem prover is used to generate test cases. A control flow graph based technique that relies on symbolic execution and a constraint solver is presented in [Yan et al., 2006]. In [Zheng et al., 2007], an encoding into Promela is used and test cases are generated using counter examples produced by the SPIN model-checker. An approach based on STS models is proposed in [Frantzen et al., 2009], where test cases are built on-the-fly by simulating the specification step by step. The models used for testing are retrieved from BPEL specifications of the orchestrations in [Yan et al., 2006, Zheng et al., 2007], while [Frantzen et al., 2009] relies on UML state diagrams for this. Most of the service testing approaches supporting behavioral specifications are interested in test case generation, but do not tackle the remaining steps with fully-automatic test passing and online test oracle implementation [Rusli et al., 2011]. In [Sinha and Paradkar, 2006, Yan et al., 2006], passing the tests (including tool support for this) is not addressed. This is achieved using JUnit in [Zheng et al., 2007], still, the user has to enter input data. In [Frantzen et al., 2009], random generation is used to create input data. However, in their on-the-fly approach, input data are realized transition by transition. Hence it is not possible to select given symbolic paths to test (test objectives) and generate data for it. Combining on-the-fly testing with symbolic execution is a perspective of [Frantzen et al., 2009].

Verification of service choreographies has mainly been addressed with a design-time verification point of view, see, *e.g.*, [Busi et al., 2006, Qiu et al., 2007, Foster et al., 2010, Roohi and Salaün, 2011, Basu and Bultan, 2011], developing techniques for peer generation or checking the conformance of such peers (steps $A1$–$A2$ and B in Fig. 1.9, right). While orchestration testing has been largely addressed, at least as far as testing one port (user) is concerned, few works address the testing of service choreographies. Dynamic symbolic execution is used in [Zhou et al., 2010] in an active (hence intrusive) testing approach that considers the implementation as a white box. A passive testing approach is proposed in [Andrés et al., 2010]. Specification is given as local and global invariants rather than with a choreography specification language.

Contributions. In [IP26, IP31] (Sect. 3.2), I have proposed a symbolic framework for online conformance testing of service orchestrations, thus achieving the corresponding perspective in [Frantzen et al., 2009]. Taking inspiration on earlier works in symbolic testing [Jeannet et al., 2005, Gaston et al., 2006, Frantzen et al., 2006], this framework is based on symbolic models (STS) and symbolic execution, hence avoids the state-explosion problems due to value passing in messages exchanged between services. Following principles P5 and P6, this is also an end-to-end and tool supported approach, taking into account the retrieval of specification models from a real language (ABPEL), the passing of test cases using SOAP messages and test verdict with an online testing oracle. In [IP26], testing is performed based on structural coverage criteria. This is extended in [IP31] with symbolic test objectives, that enable one to test specific properties of interest, and the use a Satisfiability Modulo Theory (SMT) solver that supports more constraints and the rich XML based structure of service data. Taking the specification as the requirement an

end-user or another (client) service has over some reusable service, this framework can be used not only at development time but also at deployment (for requirement based service selection) and at execution time (for service replacement).

As a complement to orchestration testing, that focuses on the "client" port of a centralized composition, I have tackled the testing of compositions in-the-large. In [IP32] (Sect. 3.3), I have proposed an approach for the conformance testing of distributed orchestrations with reference to choreography specifications. In this work I address the peculiarities of choreographies through non-intrusiveness (using passive testing), support for peers without source code being available (using black-box testing), and both local and global conformance. Again, this approach is fully tool supported, including the extension of a standard BPEL engine to log the SOAP messages exchanged between the distributed peers.

State of the Art and Contributions on Choreography Verification

Realizability has been initially studied for MSC [Alur et al., 2003, Uchitel et al., 2004, Alur et al., 2005], *i.e.*, interconnected interface models. It has later been studied in the context of Web services with interaction models such as specific choreography process algebras or UML collaboration diagrams. Several choreography approaches enforce realizability restricting the choreography specifications with well-formedness rules or sufficient conditions [Carbone et al., 2007, Fu et al., 2004, Bultan and Fu, 2008, OMG, 2011]. Others, like [Qiu et al., 2007], introduce new constructs and associated projections in choreography languages that result in the adding of additional messages in peers for this. The adding of messages to ensure realizability is also proposed as a solution in [Salaün and Bultan, 2009] and also relates to orchestration distribution [Gowri Nanda et al., 2004, Chafle et al., 2005] or distributed adaptation [Inverardi et al., 2005, Autili et al., 2008].

Most of the work dedicated to realizability assumes a synchronous communication model, *e.g.*, [Busi et al., 2006, Li et al., 2007, Carbone et al., 2007, Qiu et al., 2007]. Few approaches support asynchronous communication. This is achieved using sufficient conditions in [Fu et al., 2004, Bultan and Fu, 2008], bounded asynchronous communication in [Salaün and Bultan, 2009], controllability analysis in [Lohmann and Wolf, 2010], and synchronizability in [Basu and Bultan, 2011],

Contributions. In [IP33] (Sect. 3.4), I propose an approach for the realizability checking of BPMN 2.0 choreographies. A first contribution is the transformation of BPMN 2.0 choreographies into the LOTOS NT process algebra. This paves the way for further development of formal tools dedicated to the new standard BPMN 2.0 choreography notation. Realizability can be checked both in a synchronous and in an asynchronous communication framework, which fosters applicability. Finally, this approach is totally tool-supported, bridging design in the Eclipse BPMN 2.0 editor and verification using the CADP toolbox.

State of the Art and Contributions on Run-Time Repair

We have seen in Chapter 2 that software architects and end-users could benefit from automatic service composition techniques. Still, these techniques should support the automatic evolution of compositions, taking into account changes both in the needs of the (possibly mobile) end-users and in service availability, causing broken compositions.

When the issue is to react to a disappeared or faulty service, or to a service with a bad QoS, a first solution is *replacement*, *i.e.*, replacing a service by a brand new one. Replacement is an efficient technique as far as computation time is concerned. However, it is limited to 1-1 substitution. Further, it focuses on finding replacement for broken services, while solving a broken composition may require removing additional services or may introduce the need for new services in case of added requirements. Going further than 1-1 substitution, supporting both 1-n substitution and added needs, can be done with *recomposition*, *i.e.*, running a composition algorithm on an updated composition problem, hence recomputing causal input/output relations between services. Any algorithm defined for service composition [Marconi and Pistore, 2009] would apply there, including many ones based on some form of planning. Software adaptation [Seguel et al., 2008] may also be used as a repair technique in case it is possible to keep the original (broken) composition unmodified and build a mediator in between this and available services. Yet, computing a mediator is as expensive as computing a new composition.

Contributions. To correct broken compositions, I have proposed in [IP27, IP28, IP29, S3] (Sect. 3.5) *repair* as an alternative solution that goes beyond the limits of service replacement while avoiding recomposition. This technique aims not only at keeping most of the composition models as-is (not recomputing them), but also at taking benefit from them while computing a corrected composition. As such, repair is a form of *heuristic and guided partial recomposition*. In case of 1-1 substitution, repair performs as replacement and is as efficient. In other cases and for added needs, repair yields better computation time wrt. recomposition while retrieving solutions of the same quality. This approach is completely tool-equipped, including going beyond models, *i.e.*, reading ontologies and service descriptions files and generating BPEL orchestrations.

3.2 Conformance Testing of Service Orchestrations

[IP26,IP31] – work done in the context of L. Bentakouk Ph.D thesis.

Overview

In our orchestration testing framework (Fig. 3.1), the orchestration specification is first transformed into an STS model. In a second step, a SET is computed from this STS. It represents (a finite subset of) the STS execution semantics, while avoiding the usual state-explosion problem in presence of unbounded data types, as used in full-fledged BPEL. Given a coverage criterion, we generate from the SET a set of symbolic test cases which are finally realized online and run against an implementation, *i.e.*, the Service Under

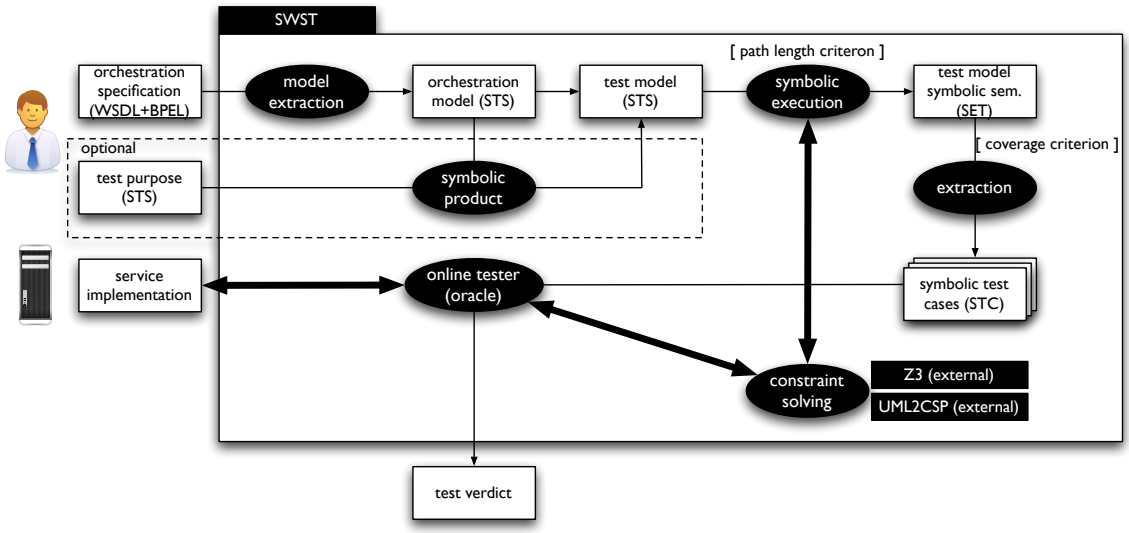

Figure 3.1: Conformance testing of service orchestrations – overview

Test (SUT). As an alternative or in combination with coverage criteria, one may describe properties of interest to be tested. This is performed using symbolic test purposes specified with STS and performing a product between the specification STS and the test purpose STS prior to SET computation. Our framework is supported by SWST (Symbolic Web Service Testing), a set of tools that we have developed. SWST relies on external constraint solvers in the SET computation (path condition consistency checking) and in the symbolic test case realization (SUT input generation and SUT outputs checking).

Models

Orchestration specification model. Several models have been proposed for service composition [ter Beek et al., 2007]. They mainly differ in their formal grounding (Petri nets, transition systems, or process algebra), and the subset of service languages being supported. We have based on [Mateescu and Rampacek, 2008] due to its wide coverage of the BPEL language constructs, which is, through extension, amenable to symbolic execution. With reference to [Mateescu and Rampacek, 2008], we support with our transformation rules (see [IP26] for details) data in computation, conditions and messages, message faults, message correlation, parallel processing (flow) and the until activity. More specifically, support for data yields grounding on (discrete time) STS and their symbolic execution, rather than on (discrete time) LTS. With reference to the BPEL to STS transformation used for adaptation in Chapter 2, for testing we rely on STS in the large, that is, also on guards and actions.

Test purposes. Test purposes (TP) enable one to focus on specific requirements, *e.g.*, functional properties, during the test process. We suppose these are given as STS too, yet, it has been shown that such transition system models could be obtained automatically

from human-friendly descriptions such as sequence diagrams [Uchitel et al., 2003]. Test purpose models are defined according to the orchestration (specification) models they refer to. Given an orchestration STS model $\mathcal{B}$, a TP for $\mathcal{B}$ is an STS TP with some constraints [IP26]. Variables and domains in TP include the $\mathcal{B}$ ones. Further, TP may introduce additional variables for expressiveness, *e.g.*, to denote additional conditions in the TP to focus on a subset of the model behavior. Events used in TP correspond to the $\mathcal{B}$ ones but for the introduction of a specific event, *. Transitions labeled with * may neither have a guard, nor actions, and are used to abstract in TP one or several $\mathcal{B}$ transitions that are not relevant for the expression of the requirement. Finally, a TP defines a specific subset of the states, *Accept*, that denotes TP satisfaction (if the implementation goes up to such a state then the test passes). Finally, we impose that TP is consistent with $\mathcal{B}$, *i.e.*, TP symbolic traces are included in $\mathcal{B}$ ones. This can be checked using symbolic execution, where we also have to check that the path condition corresponding for the TP trace implies the path condition of the $\mathcal{B}$ trace.

STS symbolic product. To generate test cases we have to take into account constraints specified both in the specification and in the TPs. This is achieved using an STS (symbolic) product that is built using four rules (see [IP31] for the detail of these rules). Two rules are used for asynchronous evolution of the model (resp. the TP) on non observable events. The third rule is use to synchronize the model and the TP on common explicited communication events. Finally, the last rule is used to synchronize the model and the TP on common implicit communication events (corresponding to $*$ in the TP).

Tests Generation and Execution

SET Computation. The SE of a program is represented by a SET. Since we apply SE to the STS obtained from orchestrations, the program counter in the SET is an STS state, and its variables correspond to the STS variables, *i.e.*, in turn, to the specification variables. In case the STS results from the product between the specification STS and a TP STS, additional variables used in the TP are also part of the SET variables. The edges of the SET, $\mathcal{E}_{\text{SET}}$, are elements of $\mathcal{N}_{\text{SET}} \times Ev_{\text{symb}} \times \mathcal{N}_{\text{SET}}$, where Ev_{symb} corresponds to the STS events with symbolic variables in place of variables. We explain in [IP29] how the SET is computed. We also explain how we cut off the inconsistent path conditions by means of a constraint solving tool and how we prune the SET with a path length criterion ($k \in \mathbb{N}$). Once the SET is computed, we have to extract the test cases and to execute them.

STC extraction. Symbolic test cases correspond to the SET paths. However, it may be relevant to test only paths leading to orchestration termination ($\surd$) even if it is not strongly required for Web services since different instances are run for each test case. Due to our k path length criterion in the SET computation, it follows that symbolic test cases have a length $n \leq k$. Notice that we can increase the value of k if we did not find errors during the execution of tests.

Online realization with a constraint solving tool. Since Web services are reactive systems, test case realization has to be performed step by step, by interacting with the SUT. This is to avoid emitting erroneous verdicts. Let us take an execution path (*i.e.*, a

symbolic test case) $pl.o?x.pl.o!y$ ending in node $\eta = (s, \pi, \sigma)$, with $\pi = v_{s_0} > 2 \wedge v_{s_1} > v_{s_0}$ and $\sigma = \{x \rightarrow v_{s_0}, y \rightarrow v_{s_1}\}$. Realization all-at-once would yield a realized path (*i.e.*, a realized test case) $p?v_{s_0}, p!v_{s_1}$ with, *e.g.*, $\{v_{s_0} \rightarrow 3, v_{s_1} \rightarrow 4\}$. Let us suppose now that we send message p with value 3 to the SUT and that it replies with value 5. We would emit a *Fail* verdict ($5 \neq 4$), while indeed 5 would be a correct reply ($5 > 3$). Concretely, the tester is implemented as a reactive process that mirrors the observable behavior of the test case. We begin with the PC in the last state of the SET path corresponding to the test case. Whenever the tester has to send a message to the SUT, PC is solved to get correct data values to send. Whenever the tester has to receive a message from the SUT, a timeout is run. If it ends before we receive the message there is an *Inconclusive* verdict. If we receive the message from the SUT, data is extracted from it and the PC is updated with equality constraints between the reception variable(s) and the observed data, and the updated PC is then checked to see if the data is correct or not. In both cases, we rely on a constraint solving tool. Preliminary experiments have been undertaken with UML2CSP [Cabot et al., 2007] and later on with the Z3 SMT solver [de Moura and Björner, 2008], supporting more constraints in PC (hence more conditions in the BPEL specifications) and enabling more automation of the test process (no tuning of the symbolic variable domains is required).

Observability and controllability. When the tester receives an unexpected message, it does not always corresponds to an error (and hence, to a *Fail* verdict). Due to non observable events and internal communications with sub-services in orchestrations, the state where we are in the test case execution does not always correspond to the state in which a correct implementation may have evolved. Therefore, whenever we receive an unexpected message, we check if it is accepted in a set of potentially reached states. If it is not the case, we produce a *Fail* verdict, else an *Inconclusive* verdict.

3.3 Conformance Testing of Service Choreographies

[IP32] – work done in the context of H.-N. Nguyen Ph.D thesis.

Overview

Our choreography testing approach (Fig. 3.2) is as follows. At the specification level, from a choreography specification (C), we retrieve its trace set ($[\![C]\!]$), a set of local requirements (R_i) using projection, and the trace sets of these local requirements ($[\![R_i]\!]$). At the implementation level, that we denote with $Impl$, we assume a one-to-one function between the roles in C and the peers in $Impl$: each role R_i is implemented by exactly one peer P_i, and each peer P_i implements exactly one role R_i. We have a point of observation (PO) for each peer, where we gather message exchanges in a local log (l_i). Using all these logs, we synthesize a global log (L). We assume that there exists a global clock to support global log synthesis. This assumption holds, *e.g.*, when peer collaborations are deployed over clouds. Global log, L, and local logs, l_i, are then respectively checked against $[\![C]\!]$, to establish global conformance, and $[\![R_i]\!]$ to establish local conformance. Our approach is fully automated with a tool whose architecture follows Figure 3.2 and that contains

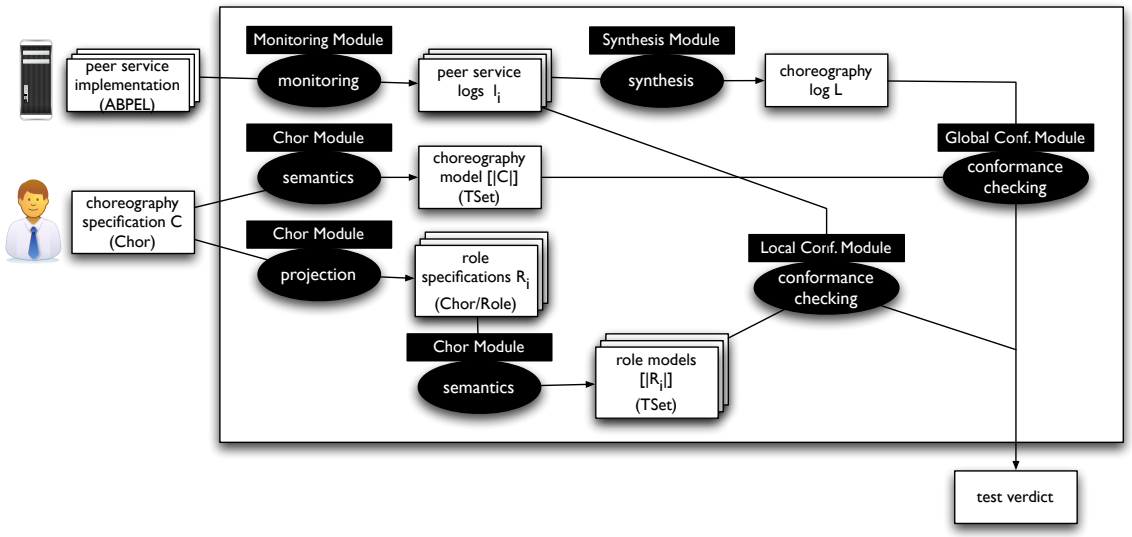

Figure 3.2: Conformance testing of service choreographies – overview

modules for SOAP message logging, global log synthesis, *Chor* management (parsing, role generation, trace semantics), global and local conformance.

Choreography specification with *Chor*

We chose the *Chor* [Qiu et al., 2007] language for the specification of choreographies. While being simpler than more general purpose languages such as BPMN or UML, it is expressive and abstract enough to enable one to specify collaborations. *Chor* can also be seen as an abstraction of the WS-CDL [W3C, 2004] standard. Further, *Chor* is based on an interaction model and defines both a choreography language, a role language, and projections between global and local (role) descriptions supporting the formal treatment of collaborations at both viewpoints.

Chor includes basic and structuring activities. *skip* denotes a do-nothing action, while $c^{[i,j]}$ represents a basic interaction on some medium c between two roles of the choreography, namely i and j, called performers of the interaction. Since *Chor* is concerned about the abstract specification of the collaboration, the instantiation of some interaction medium, and details associated to it (*e.g.*, exchanged data) is part of the developer duties. This can be, *e.g.*, using message and message parts in a Web service framework. We restrict to an observable fragment of *Chor*, *i.e.*, we do not take into account the specification at the global level of local non-observable actions. Structuring in *Chor* is achieved using sequencing (;), exclusive choice ($\sqcap$) and parallel flows ($\|$).

Role requirements are described in a dialect of *Chor* called role languages (*Role* for short) with the only difference that a global interaction $c^{[i,j]}$ corresponds in role i (resp. j) to an emission denoted $c^{[i,j]}!$ (resp. reception denoted $c^{[i,j]}?$). The semantics of a *Chor* (local or global) specification C is given in terms of its trace set [Qiu et al., 2007].

Formalization of Conformance

In passive testing, implementations may only be observed and checked for conformance using their logs, which are (linear) sequences of observations. Further, the *Chor* semantics is given in terms of its execution traces. This advocates for the use of a trace equivalence or a trace preorder. Further, the relation between a choreography specification C and an implementation $Impl$ can be seen with two mirror perspectives. In the former perspective, it is *the coordination middleware* that is tested. One is interested in the fact that $Impl$ strictly enforces (over connected peers) what is described in C. $Impl$ should then exhibit at least (or exactly) the behavior described in C. This corresponds to orchestration active testing, as shown, *e.g.*, in the previous section. In the second perspective, it is *the cooperation of the peers* that is tested. One is interested in the fact that the peers do not interact in some other ways than what is specified in C. This corresponds to passive testing using logs at the peers' locations. In such a case, we impose that the traces of $Impl$ are included in the S ones. In this work, we focus on this second perspective. We may now give the formal definition of conformance. For this we base on trace preorder.

Definition 3.3.1 (Trace preorder)**.** *Given any two traces σ_1 and σ_2, and any event α, trace preorder is defined recursively as follows: (i) $\langle\rangle \preceq \sigma_2$ and (ii) $\alpha.\sigma_1 \preceq \alpha.\sigma_2$ iff $\sigma_1 \preceq \sigma_2$.*

Given two trace sets T_1 and T_2, we write $T_1 \preceq T_2$ iff $\forall t_1 \in T_1, \exists t_2 \in T_2 \mid t_1 \preceq t_2$. Indeed, while implementing a choreography, the developer may have to add important additional exchanges or synchronizing activities in the peers. This is especially the case with non-realizable choreographies. Let us take for example the C_1 choreography specification that we saw in Chapter 1, $C_1 = {m_1}^{[P_1,P_2]}; {m_2}^{[P_3,P_4]}$. Projecting it on its roles we get $R_{P_1} = {m_1}^{[P_1,P_2]}!$, $R_{P_2} = {m_1}^{[P_1,P_2]}?$, $R_{P_3} = {m_2}^{[P_3,P_4]}!$, and $R_{P_4} = {m_2}^{[P_3,P_4]}?$. Implementing the specification using these four peers as-is, *i.e.*, $Impl = R_{P_1} \parallel R_{P_2} \parallel R_{P_3} \parallel R_{P_4}$, the developer cannot prevent that P_3 and P_4 interact on m_2 before P_1 has sent m_1 to P_2: trace $\langle {m_2}^{[P_3,P_4]}, {m_1}^{[P_1,P_2]}\rangle$ is in $[\![Impl]\!]$ while it is not in $[\![C]\!] = \{\langle {m_1}^{[P_1,P_2]}, {m_2}^{[P_3,P_4]}\rangle\}$. The developer may decide to add a synchronizing message between P_2 and P_3, m_{sync}, to enforce the choreography, *i.e.*, implementing P_2 and P_3 respectively by ${m_1}^{[P_1,P_2]}?; {m_{\text{sync}}}^{[P_2,P_3]}!$ and ${m_{\text{sync}}}^{[P_2,P_3]}?; {m_2}^{[P_3,P_4]}!$. However, in such a case, we would not have conformance, *i.e.*, $[\![Impl]\!] \not\preceq [\![C]\!]$. To support this, we formally define conformance as follows.

Definition 3.3.2 (Conformance)**.** *Given a specification S, an implementation $Impl$, and their trace sets $[\![S]\!]$ and $[\![Impl]\!]$, we have $Impl$ conf S iff $[\![Impl]\!] \downarrow_{acts(S)} \preceq [\![S]\!]$, where $acts(S)$ is the set of all activities of S and $\downarrow$ is the filter operator,* i.e.*, $t \downarrow_X$ (or $T \downarrow_X$) retains only the elements of X in t (or T) while preserving their order.*

Based on this formal definition of conformance, we may finally define choreography conformance.

Definition 3.3.3 (Choreography conformance)**.** *Given a choreography C with n roles, $R_i = nproj(C, i)$, and an implementation $Impl$ with n peers, P_i, $Impl$ conforms to C, denoted $Impl$ conf C, iff the following two conditions hold:*

i) P_i *conf* R_i*, for every* $i = 1..n$ *(*local conformance*),*

ii) $(P_1 \parallel P_2 \parallel \ldots \parallel P_n)$ *conf* C *(*global conformance*).*

Test Architecture

Using passive testing in our work, testing can be done continuously and the peers in a collaboration can evolve dynamically. Such a seamless monitoring activity is a less costly activity as it does not require to make the IUT unavailable during the testing process. Each peer is monitored at a specific PO to collect all incoming and outgoing messages. The setting up of POs, one for each peer, is guided by the ability to capture all the input and output messages of the observed peers. The captured messages that do not concern the testing choreography will be discarded using filtering (Def. 3.3.2).

Communication between peers is performed through SOAP messages. When a SOAP message is sent (or received) by a peer of the IUT, an *observation* is recorded in the log file. A *Chor* interaction $c^{[i,j]}$ means that a message c is transferred from R_i to R_j. As a consequence, an observation contains the sender, the receiver and the type of the SOAP message. To ease the reconstruction of the order between observations of different logs l_i, an observation also contains the time of the observation. In order to collect the SOAP messages exchanged between the peers of the implementation, we have extended Apache ODE, a BPEL compliant WS orchestration engine. This choice may cause a limitation by considering only the monitoring of services which are implemented in BPEL. To overcome this issue, we rely on wrappers for peers not implemented in BPEL.

3.4 Realizability Checking of Choreographies

[IP33] – work done in the context of collaborations with colleagues from INRIA Rhône-Alpes.

Overview

Our approach for checking the realizability of choreographies is described in Figure 3.3. We begin with a BPMN 2.0 choreography, described in a Choreography Diagram, that is transformed into a workflow model (see Chap. 1) where nodes correspond to BPMN 2.0 basic choreographic interactions and structuring operators. This workflow model is then encoded into LOTOS NT (LNT) [Champelovier et al., 2010], an improved version of the LOTOS value-passing process algebra with imperative programming structuring mechanisms and functional style data type definitions. We also generate encodings for the projected peers and their composition. These encodings are declarative, as opposed to an ad-hoc projection algorithm operating at the model level. From the choreography and peer composition encodings we respectively retrieve the expected choreography and the peer composition models (LTS), and finally, we check if the choreography is realizable by comparing them. In case it is not, we return a counter-example. Our approach is fully tool supported using tools that we have developed for model retrieval (BPMN2Py) and LNT encoding (Py2LNT).

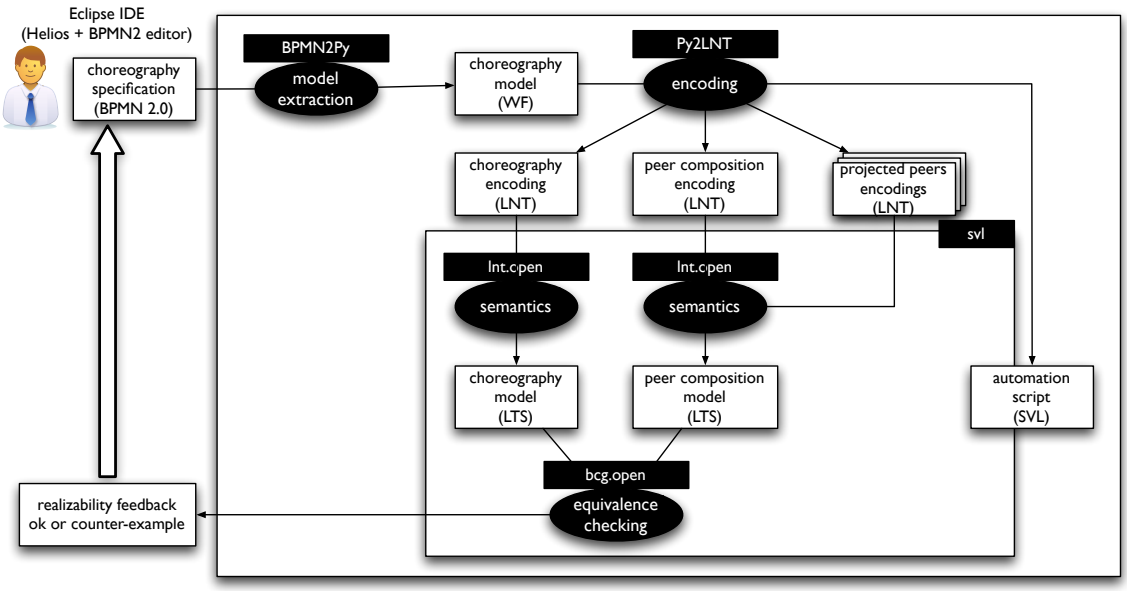

Figure 3.3: Realizability checking of choreographies – overview

At the same time than encoding, a CADP SVL script is generated that automates all further steps using CADP tools, namely lnt.open to retrieve LTSs from LNT descriptions and bcg.open to perform behavioral equivalence checking.

BPMN 2.0

The Business Process Modeling Notation, BPMN [OMG, 2011], is a visual notation that suits well the needs for the abstract specification of business processes and their interactions, including future service implementations. BPMN 1.x supported choreography specification through an interconnected interface model (Collaboration Diagrams) only. In addition to Collaboration Diagrams, BPMN 2.0 (BPMN in the sequel) introduces Choreography Diagrams to support an interaction model.

The building blocks of BPMN Choreography Diagrams are choreography tasks, one-way or two-way interactions between peers with one of them being the initiator. This corresponds closely to the abstraction of choreography specification atoms given in Chapter 1. However, in BPMN, a choreography task may have an internal marker to denote if, and how, the related interaction (one or two message exchanges) is repeated. BPMN enables to describe control flows using sequence and (diverging or converging) gateways for more complex behaviors: exclusive gateways (decision, alternative paths), inclusive gateways (inclusive decision, alternative but also parallel paths), parallel gateway (creation and merging of parallel flows), and event-based gateway (choice based on events, *i.e.*, message reception or timeout, alike BPEL pick construct).

Encoding

The encoding of a BPMN choreography ($Main$) follows a state machine pattern, where states are LNT processes that correspond to the choreography constructs, *i.e.*, nodes in the workflow model. Our encoding (see [IP33] for technical details) supports the main BPMN Choreography Diagram constructs, *i.e.*, choreography tasks, including their loop kind and value, and (diverging and converging) exclusive, parallel, and inclusive gateways.

A benefit of LNT is that it makes it easier to retrieve models for the peers and their composition. For each peer P_i in a choreography, the encoding of P_i is achieved declaratively using the hiding operator of LNT: all interactions but for the ones in which P_i is a participant are hidden in $Main$. Finally, the composition of all peers is defined as the parallel composition of the peer encodings with synchronizing on shared interactions.

Realizability Checking

Realizability checking is achieved in two steps. First, using the LNT semantics, it is possible to retrieve the LTS (explicit models) for the choreography and for the peer composition from their encodings (implicit models). In the later case, τ-reduction is performed beforehand on peer models to remove interactions hidden in their encodings. Realizability is then checked using behavioral equivalence between the choreography and the peer composition models. For this we use bisimulation and trace equivalences. As a side-effect of behavioral equivalence checking, a counter-example is returned if the choreography is not realizable.

Asynchronous communication is not directly supported by LNT. Therefore, to check for realizability in an asynchronous framework, we generate additional code for bounded FIFO buffers and accordingly we change the communication architecture of the peer composition encoding in order to make peers communicate through these buffers. The buffer size is a parameter of the encoding tool. The user may work using different sizes or check for synchronizability [Basu and Bultan, 2011] of the choreography. In the later case, verification can be done on the synchronous version of the system and the results hold for the asynchronous case.

3.5 Repair of Service Orchestrations

[IP27,IP28,IP29,S3] – work done in the context of collaborations with colleagues from Concordia University, Canada.

Overview

Our repair technique is integrated in a larger framework for composition using planning that we have implemented in Java, PGA (Planning Graph with Adaptation, Fig. 3.4). PGA includes a composition module (Java implementation of the standard graph planning algorithm [Blum and Furst, 1997]), used to build initial compositions and perform recomposition, a change module, used to apply change on a composition possibly breaking it, and a repair module, used to perform repair on broken compositions. PGA can read

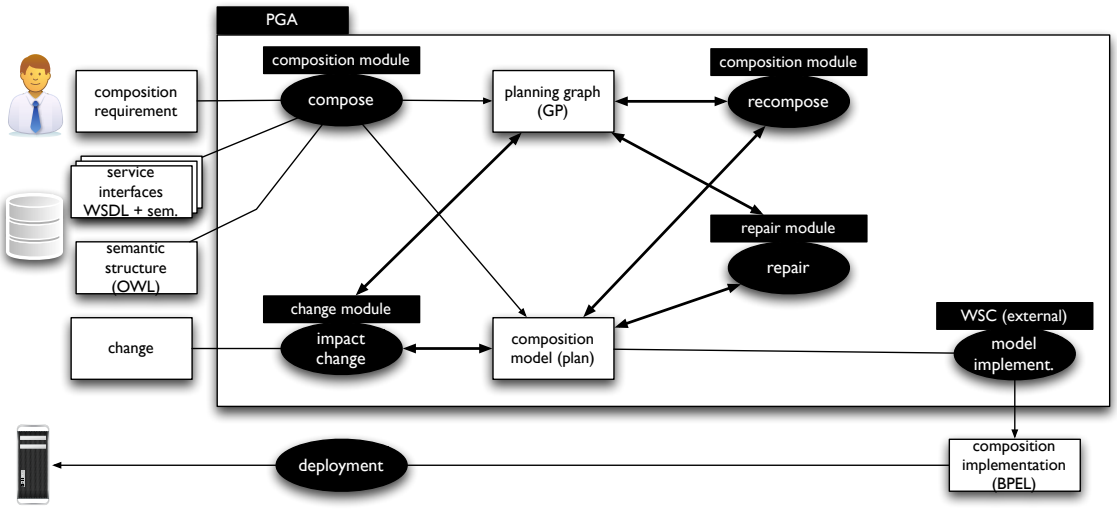

Figure 3.4: Repair of service orchestrations – overview

an OWL file describing ontology concepts and a set of annotated WSDL interfaces (we rely on simple extension mechanisms instead of full-fledged SAWSDL [W3C, 2007] for this). PGA is also connected to the WSC platform[2] which is used to generate BPEL orchestrations from XML descriptions of compositions. For the time being, PGA does not include pycompose (Sec. 2.5). This is planned as a future extension of the framework. We have applied PGA on big-size benchmarks generated by the WSC testset generator, a tool that enables to compare service composition algorithms by generating sets of semantically annotated service interfaces. The experiments have shown that using repair we get results of the same quality than with recomposition (in length, number of used services, and distance wrt. the original composition before change) but in a shorter computation time.

Models

We build on service composition using planning graphs, which, in absence of conversations, can be modeled as a tuple $(W, D_{\text{in}}, D_{\text{out}})$, where W is a set of services, D_{in} are provided inputs, and D_{out} are expected outputs. Services correspond in to planning actions A where for each a in A, we use $in(a)$ to denote the inputs of a (*i.e.*, $pre(a)$ in planning theory, Chap. 1) and $out(a)$ to denote the outputs of a (*i.e.*, $effect^{+}(a)$ in planning theory, Chap. 1). As a result of the composition process, we get a planning graph G (with its alternating proposition layers P_i and actions layers A_i) and a selected plan π. Service composition should be considered in a world that is subject to change. Hence, whether it is due to services appearing or disappearing (W'), to change in the given inputs (D'_{in}), or to new goals or goals getting out of interest (D'_{out}), we get a new composition problem $(W', D'_{\text{in}}, D'_{\text{out}})$.

[2] `http://ws-challenge.georgetown.edu/wsc09/software.html`

After impacting change, we get a partial planning graph G with a set of *Broken Preconditions* $BP^G_{m\in[1,n]}, BP^G_m \subseteq P_i$. BP^G_n corresponds to unsatisfied goals, while $BP^G_{m\in[1,n-1]}$ are inputs of actions in A_{m+1} that are no longer available. Let A be a set of actions, we denote $out(A)$ (resp. $in(A)$) the set $\bigcup_{a\in A} out(a)$ (resp. the set $\bigcup_{a\in A} in(a)$). We have $BP^G_{m\in[1,n-1]} = \{p \in in(A_{m+1}) | p \notin D_{\text{in}} \bigcup_{k\in[1,m]} out(A_k)\}$ and $BP^G_n = \{p \in D_{\text{out}} | p \notin D_{\text{in}} \bigcup_{k\in[1,n]} out(A_k)\}$. We may also focus on a given plan, say π. Let π_i be the set of actions in π at step m. Due to π computation with the planning graph we have $\pi_m \subseteq A_m$. We have then broken preconditions related to π, $BP^\pi_{m\in[1,n-1]} = \{p \in in(\pi_{m+1}) | p \not\subseteq D_{\text{in}} \bigcup_{k\in[1,m]} out(\pi_k)\}$ and $BP^\pi_n = \{p \in D_{\text{out}} | p \notin D_{\text{in}} \bigcup_{k\in[1,n]} out(\pi_k)\}$. Note that BP^G_m and BP^π_m are incomparable.

Repair Principles

The planning graph for the initial composition problem, even if broken, still contains parts that are valid for the new composition problem. Of course, if π is not broken ($\forall m \in [1,n]$, $BP^\pi_m = \emptyset$) we have nothing to do. π is still a valid solution, even in a broken planning graph. Else, we run our algorithm for BP^π (hence, in the sequel, we will use BP_m for BP^π_m for simplicity). Note that if π is broken but G is not ($\forall m \in [1,n]$, $BP^G_m = \emptyset$) then there is still at least a solution in G, which we may retrieve using backtracking. However, this may yield a solution which is very different from π. Running our algorithm we try first to get a resembling solution. If this fails, any other solution will be found too by the algorithm.

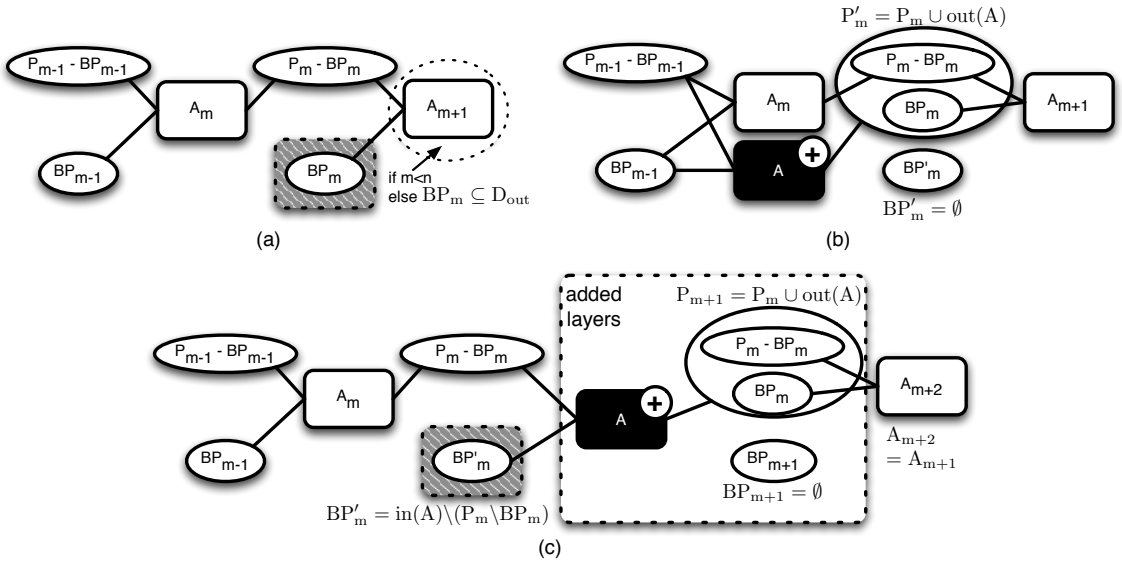

Figure 3.5: Broken preconditions at level m, $m \leq n$ (basic repair)

For a proposition level where BP_m is not empty (Fig. 3.5 (a)), we search for a set of candidate services A which can produce BP_m and insert them into action level m (Fig. 3.5

(b)). This promotes shorter repair solutions. Sometimes, the lower proposition level P_{m-1} does not contain all the inputs needed by A. We then insert A into a new action level $m+1$ (Fig. 3.5 (c)). By doing this, we can use more propositions since $P_{m-1} \subseteq P_m$ but increase the plan length by one.

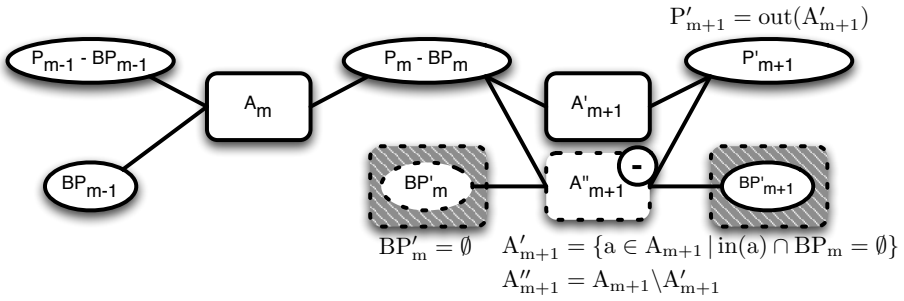

Figure 3.6: Broken preconditions at level m, $m < n$ (degraded repair)

This technique can fail if we cannot find a set of services A that produce all the broken preconditions in BP_m. As a solution we may degrade our basic repair principle. If it is not the goal layer ($m < n$), we remove the unrepairable part (Fig. 3.6). New broken preconditions may appear at level $m+1$ and will be treated in a next iteration. If it is the goal layer ($m = n$), then there is no solution, neither with repair nor with recomposition. Degraded repair increases computation time wrt. basic repair since it gets closer to recomposition. Still, it enables to find a solution whenever it exists.

Repair is a greedy search algorithm. We use two heuristic functions, (3.1) and (3.2), to select the best services. The evaluation depends on the layer where w is to be added. At the highest level n, we define:

$$f1(w,G,m) = \frac{|BP_n \cap out(w)|}{\max_\Omega} \times 2 + \frac{|in(w) \cap P_{m-1}|}{\max_{Pi}} - \frac{|in(w) - P_{m-1}|}{\max_{Pe}}, \text{if } m = n; \quad (3.1)$$

where $|BP_n \cap out(w)|$ is the number of unimplemented goals that can be implemented by w; $|in(w) \cap P_{m-1}|$ is the number of w inputs that can be provided by propositions in P_{m-1}; $|in(w) - P_{m-1}|$ is the number of w inputs that cannot be provided by propositions in P_{m-1}. This set needs to be added into BP if w is added. $\max_\Omega$, $\max_{Pi}$, and $\max_{Pe}$ are the maximum value of the nominators in the neighborhood respectively to normalize each term. The first term is given a weight of 2 to increase its significance.

If w is to be added in another layer ($m < n$) then the evaluation function is:

$$f2(w,G,m) = f1(w,G,m) + \frac{\sum_{i\in[m,n]} |BP_i \cap out(w)|}{\max_{BP}}, \text{if } m < n; \quad (3.2)$$

Compared to (3.1), the added term ($\sum_{i\in[m,n]} |BP_i \cap out(w)|$) is the number of the broken propositions in the level P_m and above that can be satisfied by w outputs. $\max_{BP}$ is the maximum value in the neighborhood to normalize each term. To ensure repair termination,

we do not select services that would reproduce the same set of broken preconditions, nor do we select a service that has been removed in degraded mode.

CHAPTER 4

Conclusions and Perspectives

In this document, I have presented my recent contributions to composite systems development, focusing on in-the-large development, since I believe that it is at the same time the most complex and added value activity in software construction. Through techniques for the automatic construction, adaptation, testing, and repair of compositions, I try to support a vision where software is more dynamic, being built on-demand from end-user requirements and modified during execution to support change in context and needs. In order to foster the automation and the applicability of the proposed techniques, I rely both on formal models and on their connection to real-life interface and implementation languages, using a layered formal model transformation approach. Further, most of the techniques presented here have been applied on the state-of-the-art composite system infrastructure, namely, (Web) services.

Looking back on the contribution lecture grid that has been given in the Introduction of this document (given again in Tab. 4.1), one may see, in practice, the application of the principle stating that there is no universal formal model. Experience with different models has brought benefits and drawbacks.

Transition systems (LTSs) are a very simple model that benefits from important tool support. Their force is in their simplicity, and they perfectly suit the needs for abstract components and service behavioral interface descriptions. Through their *symbolic extensions* (STSs), they are amenable to the modeling of more expressive interfaces descriptions and to orchestration specification, supporting data and complex related conditions. However, whenever it comes to the implementation of software entities from transition system models, their intrinsic implicit (interleaving) parallelism falls short. *Process algebras* (PAs) are mainly used as a frontend to transition systems, thanks to their expressive composition operators. As a true concurrence model, *event structures* (LESs) ease the implementation of models in languages with explicit parallelism, such as BPEL. However, event structures clearly lack an important tool support with reference to mainstream models (LTSs, PNs).

Table 4.1: Contributions - lecture grid

	issue	DL^{in}	DM^{in}	WM	DM^{out}	DL^{out}
adaptation and composition						
2.2	component adaptation	WWF	LTS	LTS	LTS	WWF
			LTS	PN	LTS	
2.3	orchestration adaptation	WSDL+BPEL	STS	PA	STS	BPEL
2.4	choreography adaptation	SAWSDL+BPEL	LTS	LTS	LTS	
		WF	LES	PN	LES	
2.5	orchestration composition	WF	GP	GP	GP	
verification and repair						
3.2	orchestration testing	ABPEL	STS	SET	STC	SOAP
3.3	choreography testing	Chor+SOAP logs	TSet	TSet	n/a	n/a
3.4	choreography checking	BPMN 2.0	WF	PA, LTS	n/a	n/a
3.5	orchestration repair	WSDL+OWL(+GP)	GP	GP	GP	BPEL

One force of *Petri nets* (PNs) is the strong relation they have with workflow languages. Further, Petri nets support natively both an interaction-vision and a resource-oriented vision of components and services, while transition systems would do indirectly (through resource reification into LTSs or through data extension). With their different semantics, Petri nets can also play a central role in between other models (LTSs, LESs). Petri nets have as many extensions as transition systems, including data extensions (*e.g.*, colored Petri nets). Finally, *graph planning* (GP). It is somehow on a different level. It does not play the model role (the propositional world and the set of actions operating on this world do). It is rather a technique whose force is in building, at once, all the solutions to some problem (up to some length). Using this property, it was possible to develop composition repair as an alternative to complete recomposition.

In the sequel, I discuss some perspectives of my work.

BPMN 2.0 choreography verification framework. Important issues in choreography-oriented design are realizability and conformance. These issues have been well studied for MSC and UML collaboration diagrams. One of my objectives is to study them on a more expressive language, the new BPMN 2.0 standard [OMG, 2011]. A first step has been done with a tool-supported technique for realizability checking [IP33] (Sect. 3.4). A perspective is to *extend the subset of BPMN choreographies taken into account*, with hierarchical structuring aspects (sub-choreographies) and, as a consequence, possibly taking into account *realizability checking in a compositional way*. A strong notion of realizability, based on behavioral equivalence, has been used in [IP33]. A second perspective is to apply *looser realizability notions* [Kazhamiakin and Pistore, 2006] based on pre-orders and partial-orders.

Composition and adaptation in a true concurrency framework. In a recent work [S2] (Sect. 2.4), I have used LES for (distributed) service composition and adaptation. These models support a true concurrency semantics and ease adaptor and orchestrator

distribution and implementation. However, a limit is that service and requirement conversations may not present cyclic behaviors. Further, LES are very simple models as far as interaction description is concerned. Occurrence nets, resulting from Petri net unfolding [Esparza et al., 2002, Hayman and Winskel, 2008], enable to benefit from Petri net expressiveness while avoiding state explosion issues resulting from cyclic behaviors. A perspective is therefore to study the application of *Petri net unfolding to service composition and adaptation in a true concurrency framework*. Recent work has addressed the combination between Petri net unfolding and planning [Hickmott et al., 2007, Bonet et al., 2008], hence could serve as a basis, at least for the composition process.

Composition and adaptation at the signature, behavior, and semantic levels. We have seen in Section 2.1 that behavioral and signature adaptation are converging into composed adaptation approaches. Still, the integration with expressive algebraic semantic adaptation approaches [Morel and Alexander, 2004, Hemer, 2005] is to achieve. A first perspective concerns the combination of these approaches in order to achieve *composition and adaptation at the signature, behavior, and semantic levels*. In a possible combination with the perspectives presented above, this could yield using data extensions of Petri nets, such as colored Petri nets [Jensen et al., 2007], for composition and adaptation. The same kind of symbolic techniques that I used for testing in [IP26,IP31] (Sect. 3.2) could be used here to avoid both over-approximation and state explosion issues. However, following a behavioral type perspective, another perspective is the study of composition and adaptation through the *embedding of composition/adaptation processes and service descriptions – including signatures, behaviors, and semantics – into SMT solvers or automated theorem provers*.

Towards eternal peer composition. In new nomad uses, collaborations between peers (choreographies) are to evolve dynamically, with an impact on the peers participating in the collaboration, on global collaboration properties, and on local peer expectations. Typical application domains are social services (*i.e.*, a service market grounded on social networks) and virtual enterprises. This makes complex (and quite unrealistic) an *a priori* vision of composition and test of compositions. Following the model-at-runtime and runtime verification visions, a perspective is to study the *combination of test, diagnosis, and adaptation techniques in an online approach supporting such a dynamic vision of collaborations*. An interesting starting point is the combination of diagnosis and planning [Chanthery and Pencolé, 2009]. Additionally, in a true dynamic perspective, behavioral models may not always come for granted with the entities that compose the systems. In such a case, passive techniques for the automated retrieval of behavioral descriptions [Musaraj et al., 2010, van der Aalst, 2011, Steffen et al., 2011] should be used. Due to the above mentioned perspectives on true concurrency and support for rich data and semantic descriptions, a perspective concerns the *combination of behavioral model retrieval techniques* such as [Esparza et al., 2010] for Petri nets (and concurrency) and [Jonsson, 2011] for behavioral models extended with data. This work has begun in the context of H. N. Nguyen PhD thesis.

Resource-centric *vs.* message-centric composition. My work focused on functional service composition in order to achieve some designer or end-user requirements. The

underlying infrastructure is mainly the Web service one, as proposed by W3C. Still, new domains emerge with an important end-user need for automatic / helped composition in a distributed context: personal information management in relation with e-governance, open data access and composition (as promoted by important public actors such as the EC), use of social networks for the exchange and composition of human services, scientific workflows, etc. A major element in these frameworks is the importance of data in compositions. At the infrastructure level, Web services (RPC/SOAP) appear to be too heavyweight. Major actors (such as Amazon or Google) hence promote the REST architecture, more lightweight and adapted to data composition. If there are actually languages for data service composition (mashup languages such as Yahoo! pipes), the relative development processes are still mainly manual. A perspective is to study *verification, composition, and adaptation techniques with a data/resource-centric perspective* and target the REST architecture. This work has begun in the context of the ANR PIMI project and R. Khefifi PhD thesis.

CHAPTER 5

Publications

I give here the list of my publications (journal articles, book chapters, conference and workshop papers). Most of them are available online:
`http://www.lri.fr/~poizat/publications_year.html`.
Selection rates are indicated when known. The publications with an underlined index are the ones selected to be given in Appendix to this document. National publications, technical reports, and editions of special issues or proceedings can be found in the above-mentioned Web page.

Journal Articles

A6 Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. *IEEE Transactions on Software Engineering*, xx(x):xxx–xxx. IEEE Comp. Soc., 2011. (**to appear, available in the journal online PrePrint section**).

A5 Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563. IEEE Comp. Soc., 2008.

A4 Christian Attiogbé, Pascal Poizat, and Gwen Salaün. A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes. *IEEE Transactions on Software Engineering*, 33(3):157–170. IEEE Comp. Soc., 2007.

A3 Pascal Poizat and Jean-Claude Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. *Journal of Universal Computer Science (J.UCS)*, 12(12):1741–1782. Springer, 2006.

A2 Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software Adaptation. *L'objet*, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities. *introduction by the guest-editors*

A1 Christine Choppy, Pascal Poizat, and Jean-Claude Royer. The Korrigan Environment. Journal of Universal Computer Science (J.UCS), 7(1):19-36. Special Issue on Tools for System Design and Verification, 2001

Book Chapters

BC6 Pascal Poizat and Thomas Vergnaud. *Distributed Systems: (vol. 2) Models and Analysis*, chapter 5 - Architecture Description Languages. ISTE, Wiley, 2011.

BC5 Pascal Poizat. *Software Specification Methods: an Overview Using a Case Study*, chapter 12 - SDL. ISTE, Hermes Science Publishing, 2006. *new edition of [BC1]*

BC4 Marc Frappier, Henri Habrias, and Pascal Poizat. *Software Specification Methods: an Overview Using a Case Study*, chapter 19 - A Comparison of the Specification Methods. ISTE, Hermes Science Publishing, 2006.

BC3 Henri Habrias, Pascal Poizat, and Marc Frappier. *Software Specification Methods: an Overview Using a Case Study*, chapter 20 - Glossary. ISTE, Hermes Science Publishing, 2006.

BC2 Pascal Poizat and Thomas Vergnaud. *Méthodes formelles pour les systèmes répartis et coopératifs*, chapter 5 - Langages de description d'architecture. Hermes, Lavoisier, 2006.

BC1 Pascal Poizat. *Software Specification Methods. An Overview Using a Case Study*, chapter 9 - SDL: a Language based on Extended Finite State Machines with Abstract Data Types. Formal Approaches to Computing and Information Technology (FACIT), Springer. out of print. 2000.

Conference and Workshop Proceedings

IP33 Pascal Poizat and Gwen Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proceedings of the ACM Symposium on Applied Computing (SAC 12)*. ACM, 2012. (**to appear**).

IP32 Huu Nghia Nguyen, Pascal Poizat and Fatiha Zaïdi. Passive Conformance Testing of Service Choreographies. In *Proceedings of the ACM Symposium on Applied Computing (SAC 12)*. ACM, 2012. (**to appear**).

IP31 Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. Checking the Behavioral Conformance of Web Services with Symbolic Testing and an SMT Solver. In *Proceedings of the International Conference On Test and Proofs (TAP 11)*, volume 6706 of *Lecture Notes in Computer Science*, pages 33–50. Springer, 2011.

IP30 Pascal Poizat and Yuhong Yan. Adaptive Composition of Conversational Services through Graph Planning Encoding. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 10)*, volume 6416 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

IP29 Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repairs vs. Recomposition for Broken Service Compositions. In *Proceedings of the International Conference on Service Oriented Computing (ICSOC 10)*, volume 6470 of *Lecture Notes in Computer Science*, pages 152–166. Springer, 2010. selection rate: 15%

IP28 Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Self-Adaptive Service Composition through Graphplan Repair. In *Proceedings of the IEEE International Conference on Web Services (ICWS 10)*, pages 624–627, 2010. *work in progress track*

IP27 Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repairing Service Compositions in a Changing World. In selected papers from the 8th ACIS conference on Software Engineering Research, Management & Applications (SERA 10), volume 296 of *Studies in Computational Intelligence*, pages 17–36. Springer, 2010. selection rate: 14%

IP26 Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems.. In *Proceedings of the 21th IFIP International Conference on Testing of Communicating Systems (TESTCOM'09)*, volume 5826 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2009. selection rate: 35%

IP25 Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In *Proceedings of the International Conference on Service Oriented Computing (ICSOC 08)*, volume 5364 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2008. selection rate: 21%

IP24 Sandrine Beauche and Pascal Poizat. Automated Service Composition with Adaptive Planning. In *Proceedings of the International Conference on Service Oriented Computing (ICSOC 08)*, volume 5364 of *Lecture Notes in Computer Science*, pages 530–537. Springer, 2008. *short papers* selection rate: 34%

IP23 Tarek Melliti, Pascal Poizat, and Sonia Ben Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE 08)*, volume 4961 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2008. selection rate: 26%

IP22 Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel, and Pascal Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS'07)*, Electronic Notes in Theoretical Computer Science, volume 215, pages 39–55, 2008.

IP21 Radu Mateescu, Pascal Poizat, and Gwen Salaün. Behavioral Adaptation of Component Compositions based on Process Algebra Encodings. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 07)*, pages 385–388. ACM,IEEE, 2007. *short papers* selection rate: 24%

IP20 Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel, and Pascal Poizat. Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0. In *Proceedings of the Twelfth International Workshop on Component-Oriented Programming (WCOP'07)*, pages 9–13, 2008.

IP19 Serge Haddad and Pascal Poizat. Transactional Reduction of Component Compositions. In *Proceedings of the IFIP International Conference on Formal Methods for Networked and Distributed Systems (FORTE 07)*, volume 4574 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 2007. selection rate: 33%

IP18 Pascal Poizat and Gwen Salaün. Adaptation of Open Component-based Systems. In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2007. selection rate: 38%

IP17 Pascal Poizat, Gwen Salaün, and Massimo Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS'06)*, Electronic Notes in Theoretical Computer Science, volume 182, pages 155–170, 2006. selection rate: 62%

IP16 Pascal Poizat, Gwen Salaün, and Jean-Claude Royer. Bounded Analysis and Decomposition for Behavioural Descriptions of Components In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2006. selection rate: 31%

IP15 Carlos Canal, Pascal Poizat, and Gwen Salaün. Synchronizing Behavioural Mismatch in Software Composition In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2006. selection rate: 31%

IP14 Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer A Java Implementation of a Component Model with Explicit Symbolic Protocols In *Proceedings of the International Workshop on Software Composition (SC'05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 115–124. Springer, 2006. selection rate: 32%

IP13 Gwen Salaün and Pascal Poizat. Interacting Extended State Diagrams. In *Proceedings of the International Workshop on Semantic Foundations of Engineering Design Languages (SFEDL'04)*, Electronic Notes in Theoretical Computer Science, 115:49-57. 2005 selection rate: 66%

IP12 Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking Asynchronously Communicating Components using Symbolic Transition Systems. In *Proceedings of Distributed Objects and Applications. On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences (DOA'04)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer, 2004. selection rate: 25%

IP11 Marc Aiguier, Fabrice Barbier, and Pascal Poizat. A Logic with Temporal Glue for Mixed Specifications. In *Proceedings of the International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03)*, Electronic Notes in Theoretical Computer Science, 97:155-174. 2004

IP10 Christian Attiogbé, Pascal Poizat, and Gwen Salaün. Specification of a Gas Station using a Formalism integrating Formal Datatypes within State Diagrams. In *Proceedings of the International Conference on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)*. IEEE Computer Society Press. 2003

IP9 Christian Attiogbé, Pascal Poizat, and Gwen Salaün. Integration of Formal Datatypes within State Diagrams. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'03)*, volume 2621 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 2003. selection rate: 22%

IP8 Michel Allemand, Christian Attiogbé, Pascal Poizat, Jean-Claude Royer, and Gwen Salaün. SHE'S Project: a Report of Joint Works on the Integration of Formal Specification Techniques. In *Proceedings of the International Workshop on Integration of Specification Techniques with Applications in Engineering (INT'02)*. 2002

IP7 Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Formal Specification of Mixed Components with Korrigan. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'01)*. IEEE Computer Society Press. pages 169–176. 2001 selection rate: 30%

IP6 Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Specification of Mixed Systems in Korrigan with the Support of an UML-Inspired Graphical Notation. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'01)*,

volume 2029 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2001 selection rate: 30%

IP5 Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. In *Proceedings of the International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2000. selection rate: 55%

IP4 Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Integration and Composition of Static and Dynamic "Views": Unifying Approach to Complex System Specification. In *Proceedings of the Workshop on Integration of Specification Techniques with Applications in Engineering (INT'00)*, Technische Universitat Berlin, Bericht-Nr. 2000/04, ISSN 1436-9915, pages 12-20. 2000

IP3 Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: A Concurrent Automata Approach. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 939-962. Springer, 1999. selection rate: 36%

IP2 Henri Habrias, Pascal Poizat, and Jean-Yves Lafaye. A Study of Collaborative Work: Answers to a Test on Formal Specification in B. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 709 of *Lecture Notes in Computer Science*,pages 1856-1857. Springer, 1999. *long version (16p) presented at ZUM'98 Education Session.*

IP1 Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In *Selected Papers of the 13th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'98)*, volume 1589 of em Lecture Notes in Computer Science, pages 276–291. Springer, 1999. selection rate: 49% from the papers selected for the workshop

Others

S3 Yuhong Yan, Pascal Poizat and Min Chen. Repair of Service Compositions as an Adaptation to Change. submitted (journal).

S2 Pascal Poizat and Paraskevi Zerva. Distributed Service Composition with Event Structures. to be submitted.

S1 Rania Khefifi, Pascal Poizat and Fatiha Saïs. PI2M: a Flexible and Contextual Personal Information Management System. to be submitted.

CHAPTER 6

Curriculum Vitæ

Situation and History

Since September, 2001, I am an associate professor[1] at the University of Evry Val d'Essonne (Paris Metropolitan Area), France. I first developed my research activity within the IBISC (ex-LaMI) laboratory, working on topics related to composite system design and verification. During two years, 2006–2008, I was an invited researcher in the ARLES project-team at INRIA Paris-Rocquencourt, France. The expertise of this team is both on formal aspects related to software architectures, and on their relation with and implementation in efficient middleware. At this occasion, I directed my research activity towards issues related to the actual implementation of software architectures, namely Web services. In this field, I tackled how formal methods could help, in practice, both designers and end-users, in their service orchestration duties. This led to proposals for software adaptation and automated composition (Chap. 2). In September, 2008, going back from INRIA, I moved to the LRI laboratory, more precisely in the Formal Testing and System Exploration (ForTesSE) team, whose expertise is on model-based testing, with a recent focus on testing distributed systems. I complemented at this occasion the cycle of composite system development, namely with testing and repair techniques for service composition (Chap. 3).

Academic Positions

- 1997– 2000, PhD with national grant (moniteur), University of Nantes ;
- 2000–2001, temporary assistant professor (ATER 100%), University of Nantes ;
- since 2001, associate professor, University of Evry Val d'Essonne.

[1] This means half-time teaching and half-time doing research. Some administrative duties (managing a cursus) are also part of an associate professor's work.

Mobility

I did a mobility between my PhD and my associate professor position. During 2006–2008, I was on-the-move ("délégation") in the ARLES project-team at INRIA. In September, 2008, my research affiliation changed from IBISC (Evry) to LRI (Orsay). I have also been invited for short periods (one week) at the Universities of Extremadure (2005) and Málaga (2005, 2007, 2009), Spain.

Diplomas and Titles

- Sept. 1997 – Dec. 2000 : *PhD in Computer Science* (mention "très honorable")
  KORRIGAN : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes (KORRIGAN: a formalism and a method for the structured formal specification of mixed systems)
  defended on December 18th, 2000, in front of the jury : Frédéric Benhamou (president, Univ. of Nantes, France), Maritta Heisel (reporter, Univ. of Magdebourg, Germany), Gianna Reggio (reporter, Univ. of Genes, Italy), Michel Lemoine (examiner, ONERA Toulouse, France), Christine Choppy (PhD director, Univ. Paris 13, France), Jean-Claude Royer (co-advisor, Univ. of Nantes, France)
  prepared at the University of Nantes, IRIN (now LINA) laboratory ;

- 1995 : *Research Master (2nd year) in Computer Science* (ranked 1st, mention "bien")
  Applications de la réécriture de termes aux modèles à objets (Applications of term rewriting to object models)
  prepared at the University of Nantes, IRIN (now LINA) laboratory ;

- 1994 : *Master (1st year) in Computer Science* (mention "assez bien") - University of Nantes ;

- 1993 : *Licence in Computer Science* (mention "assez bien") - University of Nantes ;

- 1992 : *DUT Informatique* - University of Nantes ;

- 1990 : *Baccalauréat, série C* - Lycée Albert Camus, Nantes.

Impact of Research on Teaching

Since 2001, but for two years while being on-the-move at INRIA, I have been doing teaching at different levels (Licence, Master) and for both professional-oriented students and research-oriented ones.

My research activities have an important impact on my teaching ones. Courses on design methods and languages such as Merise or the UML, are now usual in most software engineering cursus. Basing on my experience in relating formal methods and software engineering, I had the occasion to introduce some flavor of automated verification in courses on object-oriented design and the on design and development of distributed applications, in the first and second year of Master. In this context, I have shown to the students the

impact of erroneous design in modern applications where distribution, communication, and interaction play an important part. I have then introduced a basic behavioral model (LTS), different parallelism and communication semantics, and tools for equivalence and deadlock-freeness checking (CWB, LTSA, CADP). The other way round, students regularly participated to my research activities, through PhD and Master internships (see below) but also in earlier years (summer internships in third year of Licence, introduction to research practical work in the first year of Master), through participation in the development of research prototypes.

Formal methods have now reached a maturity level for being integrated within software engineering courses. Still, this should be as transparent and automated as possible. This is clearly the case for static (data and function related) formal methods based on program assertions and testing, with tools such as Spec# [Barnett et al., 2011]. I believe that in the next years a challenge is to leverage dynamic (behavior related) formal methods in the same way. Transparent background tool support and relation to standard models and languages is a cornerstone for this.

Administrative Activities

Cursus responsibilities (University of Evry Val d'Essonne)

- director of the third year of licence in computer science, MIAGE cursus (since 2009) ;
- director of the third year of licence in computer science, INFO cursus (2008–2009) ;
- director of the second year of master, complementary competences in computer science (2004–2006) ;
- deputy director of the second year of master, complementary competences in computer science (2001–2004).

Recruiting committees and scientific councils

- external member of the recruiting committee for computer science, University of Nantes (one associate professor position in 2009) ;
- member of the recruiting committee for computer science, University of Evry Val d'Essonne (temporary assistant professor positions in 2008–2011) ;
- external member of the recruiting committee for computer science, CNAM Paris (associate professor positions in 2004-2007) ;
- elected member of the recruiting committee for computer science, University of Evry Val d'Essonne (associate professor positions in 2004-2008) ;
- elected member of the IBISC laboratory council (2006-2008).

Technical commissions

- member of the hardware/software commission, LRI lab. (since 2008) ;
- member of the documentary commission, INRIA Paris-Rocquencourt (2006–2008) ;
- member of the Web commission, LaMI/IBISC lab. (2001–2006) ;
- webmaster of the LaMI/IBISC lab (2004–2008).

Research Activities

Editions of Journal Special Issues

- **co-editor** of the special issue "*Coordination and Adaptation Techniques*"
  L'Objet, volume 12, number 1, 2006 ;
- **co-editor** of the special issue "*Software Adaptation*"
  Journal of Universal Computer Science (J.UCS, Springer), volume 14, number 13, 2008 ;
- **co-editor** of the special issue on the FOCLASA 2007 workshop
  Science of Computer Programming (SCP, Elsevier), volume 76, number 1, 2011 ;
- **co-editor** of the special issue on the FOCLASA 2008 workshop
  Science of Computer Programming (SCP, Elsevier), volume 76, number 8, 2011.

Lecture Boards for Journal Special Issues

- **reviewer** (2008) for the special issue on FACS'08
  Science of Computer Programming (SCP, Elsevier) ;
- **reviewer** (since 2005, 5 issues) for the special issues on the FOCLASA workshop
  Science of Computer Programming (SCP, Elsevier) ;
- **reviewer** (since 2005, 2 issues) for the special issues on the FSEN conference
  Fundamenta Informaticae ;
- **reviewer** (2008) for the special issue "*Architectures Logicielles*"
  L'Objet ;
- **reviewer** (2008) for the special issue "*Components, Services and Aspects: Techniques and Tools for Verification*"
  L'Objet ;
- without being in the lecture board, I regularly make reviewers for regular issues of journal articles (Computer Languages, Systems & Structures, IEEE Transactions on Automatic Control, Science of Computer Programming, Technique et Science Informatiques).

Organization and Program Committees

- **co-president of the PC, co-organizer, and PC member** (2004-2007) of the International Workshop on Coordination and Adaptation Techniques, WCAT, at ECOOP ;
- **co-president of the PC, co-organizer** (2007 and 2008) and **PC member** (2006-2010) of the International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA, at CONCUR and then at ICALP ;
- **PC member** (since 2011) of the CSI International Symposium on Computer Science and Software Engineering, CSSE ;
- **PC member** (since 2008) of the IEEE International Conference on e-Business Engineering, ICEBE (ECORE 2010 rank B) ;
- **PC member** (since 2009) of the International Conference on Software and Data Technologies, ICSOFT (ECORE 2010 rank B) ;
- **PC member** (since 2011) of the IEEE International Conference on Web Services, ICWS (ECORE 2010 rank A) ;
- **PC member** (since 2011) of the Doctoral Symposium of the International Conference on Service-Oriented Computing, ICSOC PhD Symposium ;
- **PC member** (since 2006) of Conférence nationale sur les Architectures Logicielles (National Conference on Software Architectures), CAL ;
- also **PC member** of international workshops (ProVeCS 2007, FACS 2008, SASE 2008, ORGMOD 2009, MOCA 2009, WA-SELF 2008-2009, AVYTAT 2010, WCSI 2010, CAMPUS 2010-2011, Web Service Cup 2010-2011, WS-FMDS 2011) and national workshops (OCM-SI 2005-2006).

Expertise

- **evaluation** for a national research-enterprise PhD grant (CIFRE), 2011 ;
- **evaluation** of a project for the French National Agency for Research (ANR), JCJC SIMI2 2011 call, 2011 ;
- **half-period evaluation** of a project for the French National Agency for Research (ANR), ARPEGE 2008 call, 2010 ;
- **evaluation** of a project for the French National Agency for Research (ANR), ARPEGE 2010 call, 2010 ;
- **evaluation** of a project for the Dutch Organization for Scientific Research (Nederlandse Organisatie voor Wetenschappelijk Onderzoek, NWO), 2010 ;

- **evaluation** of a project for the Dutch Organization for Scientific Research (Nederlandse Organisatie voor Wetenschappelijk Onderzoek, NWO), 2008 ;
- **evaluation** of a project for the State-Region Contractual Plan (Contrat de Plan Etat-Région Lorraine), CPER MISN 2007-2013 call, special theme on systems safety and security, 2007.

PhD Boards

- **European thesis reviewer** for the PhD of Javier Cubo Vilalba, *DAMASCO - Discovery, Adaptation and Monitoring of Context-Aware Services and Components*, University of Málaga, Spain, 2010 ;
- **European thesis reviewer** for the PhD of Aitor Urbieta, *Enfoque integrado para el modelado, emparejamiento y composición sensible al contexto de servicios semánticos, basado en precondiciones y effectos, orientado a entornos inteligentes* (Framework for the context-aware modelling, discovery, and composition of semantic services, based on preconditions and effects, for smart environments), University of Mondragón, Spain, 2010 ;
- **examiner** for the PhD of Javier Cámara Moreno, *Run-Time Behavioural Adaptation of Components and Services*, University of Málaga, Spain, 2009 ;
- **European thesis reviewer** for the PhD of Leire Etxeberria, *Evaluación de atributos de calidad en líneas de productos software de forma efectiva en costes* (Efficient evaluation of quality attributes for software product lines), University of Mondragón, Spain, 2008 ;
- **examiner** for the PhD of Miguel Ángel Pérez Toledano, *Titán: un marco de trabajo para el estudio de la integración de aspectos en sistemas software* (Titan: a framework for the study of aspect integration in software systems), University of Extremadure, Spain, 2008.

Working Groups and Collaborations

Since 2011 I co-direct with R. Rouvoy (LIFL and INRIA) the COSMAL (Components, Objects, Services: Models, Architectures and Languages) working group of the CNRS GPL (Software Engineering) national research group (GDR).

In the last years, I made collaborations with colleagues in other laboratories, and I participated to several working groups:

- CSE Department, Concordia University, Canada ;
- VASY project-team, INRIA Rhône-Alpes ;
- CNRS Software Engineering national research group (GDR GPL), COSMAL working group (member of the board, co-direction since 2011) - this working group is made up of teams in laboratories from all over France ;

- MeFoSyLoMa (Formal Methods for Software and Hardware Systems) working group - this working group is made up of teams in laboratories in the Paris metropolitan area ;
- past collaborations: DI, Università degli Studi dell'Aquila, Italy ; DIS, Università di Roma 'La Sapienza', Italy ; LCC, Universidad de Málaga, Spain ; Quercus Group, Universidad de Extremadura, Spain ; GDR ALP / AFADL and OCM working groups ; GDR I3 / OCM-SI working group ; GDR Programmation / ELO working group ; ARLES project-team, INRIA Paris-Rocquencourt, France ; IBISC, Université d'Evry Val d'Essonne, France ; LAMSADE, Université Paris Dauphine, France ; LINA, Université de Nantes, France ; LIPN, Université Paris 13, France ; OBASCO project-team, INRIA Bretagne Atlantique, France ; POP ART project-team, INRIA Rhône-Alpes, France.

Tutorials and Seminars

I have been invited to present my work at several occasions:

- P. Poizat. *Model-Based Software Adaptation. Introduction and Contributions.* (several versions)
  - invited seminar, ADAM project-team, INRIA Lille - Nord Europe, Jan. 2010 ;
  - invited seminar, Inf. Syst. Group, IEIS, Technical University of Eindhoven, Nov. 2009 ;
  - NICTA-INRIA/LAAS/LRI meeting, LRI, University Paris Sud, Jul. 2009 ;
- P. Poizat. *Testing Web Service Orchestrations.*
  - invited seminar/doctoral lecture, University of Málaga (Spain), May 2009 ;
- P. Poizat. *Model-based Software Adaptation. Open Systems & Semantics.*
  - invited seminar/doctoral lecture, University of Málaga (Spain), Mar. 2007 ;
- P. Poizat. *Adaptation Logicielle.* (several versions)
  - invited seminar, INRIA Rhône-Alpes, Grenoble, May 2006 ;
  - invited seminar, MeFoSyLoMa working group, University Paris 13, May 2006 ;
  - invited seminar, ACI FIACRE project meeting, ENST, Feb. 2006 ;
- P. Poizat. *Extension of Behaviours with Formal Datatypes.*
  - invited seminar, University of Extremadure (Spain), Jun. 2005 ;
  - invited seminar/doctoral lecture, University of Málaga (Spain), Jun. 2005.

Besides, I have been invited to make a tutorial on the use of script languages at a physics colloquium:

- P. Poizat. *Shell Scripting for Scientific Programming : the Python example.* International Workshop on Computing for Heavy Ion Physics, Apr. 2005 `http://indico.cern.ch/conferenceDisplay.py?confId=a05210`

The slides of these tutorials and seminars are available online:
`http://www.lri.fr/~poizat/communications.html`

Projects

- **ANR PIMI** – *co-writer and scientific leader for the LRI partner*
  type: French National Agency for Research, VERSO (Future Networks and Services)
  partners: Genigraph (leader), INRIA/MERLIN, IRIT/LILaC, IT Sud Paris (ex-INT), LRI/{Fortesse,IASI}, Montimage, Région Midi-Pyrénées, CTIE
  period: 36 months (label 2010, Nov. 2010 – Oct. 2013)
  goals: framework for the design and deployment of Personal Information spaces – definition of PI models and PI usage, data integration and automatic composition of e-services, trust and security.
  URL: n/a

- **ANR PERSO** – *project leader*
  type: French National Agency for Research, JCJC (Young Researchers Program)
  partners: INRIA/ARLES (leader), IBISC, LAMSADE
  period: 36 months (label 2007, Nov. 2007 – Oct. 2010)
  goals: service composition in pervasive computing – study, analysis and elaboration of a tool-equipped approach for the composition and adaptation of services, taking into account the different interface description levels (signature, protocols, quality of service, semantics) ; monitoring and runtime recomposition of service compositions.
  URL: `http://anr-perso.ibisc.univ-evry.fr/`
  (semestrial reports, half-period report, final report, poster, publications)

- **CRE SO(MP)2** – *member*
  type: externalized research contract, France Télécom R&D
  partners: LIP6 (leader), IBISC, FT R&D
  period: 8 months (May 2007 – Dec. 2007)
  goals: multi-protocol auction marketplaces – application of service-oriented architectures to auction protocols and Web service choreography adaptation

- **RNRT STACS** – *co-writer and member*
  type: National Network for Telecommunication Research
  partners: Thales Communications (leader), IBISC, CEA Saclay, Ligeron S.A.
  period: 36 months (label 2002, Dec. 2003 – Nov. 2006)
  goals: abstract and compositional specification and testing – heterogeneous specifications (UML/SDL/Esterel), integration of data into behaviors requiring the use of symbolic transition systems (STS) to avoid state space explosion

Scientific Advisory

PhD Advisory

Lina Bentakouk
Dec. 2007 – Dec. 2011
Test symbolique de services Web composites (Symbolic Testing of Composite Web Services)
advisory: 40% (direction: M.-C. Gaudel, 20%, co-advisor: F. Zaïdi, 40%)
I am co-advising this PhD since my arrival at LRI (Sept. 2008).

Huu Nghia Nguyen
Jan. 2010 – (*Jan. 2013*)
Approche de coordination distribuée à base de test et de diagnostic (Distributed Coordination Approach based on Test and Diagnosis)
advisory: 50% (direction: F. Zaïdi, 50%)

Rania Khefifi
Nov. 2010 – (*Nov. 2013*)
Model-based Techniques for Automatic PIMS Construction
advisory: 50% (direction: P. Poizat, 50%, co-advisor: F. Saïs, 50%)
I have obtained a derogation to direct this PhD without habilitation.

Master Research Internships

Paraskevi Zerva
2010 / *Automatic Mashup Composition for Sensor Data*
advisory: 80% (co-advisor: T. Melliti (IBISC, U. Evry), 20%)
Paraskevi has been selected for a PhD in Germany and another one in Ireland.

Sandrine Beauche
2007 / *Adaptation de services en informatique diffuse (Service Adaptation for Pervasive Computing)*
advisory: 50% (co-advisor: S. Ben Mokhtar (INRIA), 50%)
Sandrine is an engineer within the ARLES project-team at INRIA Paris-Rocquencourt.

Ouahiba El Gares
2004 / *Etude d'architectures de composants hétérogènes (Heterogeneous Component Architectures)*
advisory: 100%

Abdelghani Sedkaoui
2003 / *Implantation d'un modèle à composants en Java (Implementation of a Component Model in Java)*
advisory: 80% (co-advisor: J.-C. Royer (LINA, INRIA/Ecole des Mines de Nantes), 20%)

Redouane Layadi
2003 / *Analyse d'architectures de composants (Component Architecture Analysis)*
advisory: 50% (co-advisor: P. Le Gall (U. Evry), 50%)

Bibliography

[Agha, 2002] Agha, G. (2002). Special Issue on Adaptive Middleware. *Communications of the ACM*, 45(6):30–64. 17, 35

[Allen and Garlan, 1997] Allen, R. J. and Garlan, D. (1997). A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249. 11, 30

[Alur et al., 2003] Alur, R., Etessami, K., and Yannakakis, M. (2003). Inference of Message Sequence Charts. 29(7):623–633. 61

[Alur et al., 2005] Alur, R., Etessami, K., and Yannakakis, M. (2005). Realizability and Verification of MSC Graphs. *Theoretical Computer Science*, 331(1):97–114. 61

[Andersen, 1994] Andersen, H. R. (1994). Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30. 47

[Andrés et al., 2010] Andrés, C., Cambronero, M., and Núñez, M. (2010). Passive Testing of Web Services. In *Proc. of WS-FM'10*. 60

[Arnold, 1994] Arnold, A. (1994). *Finite Transition Systems - Semantics of Communicating Systems*. Prentice Hall. 24, 25, 39

[Autili et al., 2007] Autili, M., Inverardi, P., Navarra, A., and Tivoli, M. (2007). SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-based Systems. In *Proc. of ICSE'07*, pages 784–787. IEEE Computer Society. 34

[Autili et al., 2008] Autili, M., Mostarda, L., Navarra, A., and Tivoli, M. (2008). Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems. *Journal of Systems and Software*, 81(12):2210–2236. 34, 41, 61

[Baeten, 2005] Baeten, J. C. M. (2005). A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146. 26

[Balsamo et al., 2004] Balsamo, S., Marco, A. D., Inverardi, P., and Simeoni, M. (2004). Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310. 7

[Barnett et al., 2011] Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and Verification: The Spec# Experience. *Communications of the ACM*, 54(6):81–91. 87

[Barros et al., 2009] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., and Madelaine, E. (2009). Behavioural models for distributed Fractal components. *Annales des Télécommunications*, 64(1–2):25–43. 6, 7

[Basu and Bultan, 2011] Basu, S. and Bultan, T. (2011). Choreography conformance via synchronizability. In *Proc. of WWW'11*. 12, 60, 61, 70

[Becker et al., 2006] Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., and Tivoli, M. (2006). Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938, pages 193–215. 33

[Beckman et al., 2010] Beckman, N. E., Nori, A. V., Rajamani, S. K., Simmons, R. J., Tetali, S., and Thakur, A. V. (2010). Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508. 13

[Ben Mokhtar et al., 2007] Ben Mokhtar, S., Georgantas, N., and Issarny, V. (2007). COCOA: COnversation-based Service Composition in PervAsive Computing Environments with QoS Support. *Journal of Systems and Software, Special Issue on ICPS'06*, 80(12):1941–1955. 6, 37

[Benatallah et al., 2005a] Benatallah, B., Casati, F., Grigori, D., Motahari-Nezhad, H. R., and Toumani, F. (2005a). Developing adapters for web services integration. In *CAiSE*, pages 415–429. 34

[Benatallah et al., 2005b] Benatallah, B., Hacid, M.-S., Léger, A., Rey, C., and Toumani, F. (2005b). On automating web services discovery. *VLDB J.*, 14(1):84–96. 6, 14

[Benigni et al., 2007] Benigni, F., Brogi, A., and Corfini, S. (2007). Discovering Service Compositions that Feature a Desired Behaviour. In *Proc. of ICSOC'07*. 37

[Berardi et al., 2004] Berardi, D., Giacomo, G. D., Lenzerini, M., Mecella, M., and Calvanese, D. (2004). Synthesis of Underspecified Composite e-Services based on Automated Reasoning. In *Proc. of ICSOC*. 37

[Bernardo et al., 2002] Bernardo, M., Donatiello, L., and Ciancarini, P. (2002). *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459 of *Lecture Notes in Computer Science*, chapter Stochastic process algebra: From an algebraic formalism to an architectural description lamguage, pages 236–260. Springer. 7

[Bernardo and Inverardi, 2003] Bernardo, M. and Inverardi, P., editors (2003). *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*. Springer Verlag. 11

[Bernardo and Issarny, 2011] Bernardo, M. and Issarny, V., editors (2011). *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*. Springer. 20

[Bertoli et al., 2010] Bertoli, P., Pistore, M., and Traverso, P. (2010). Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3–4):316–361. 6, 37

[Bertolino et al., 2009] Bertolino, A., Inverardi, P., Pelliccione, P., and Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proc. of ESEC/FSE 2009*, pages 141–150. 6

[Blum and Furst, 1997] Blum, A. L. and Furst, M. L. (1997). Fast Planning through Planning Graph Analysis. *Artificial Intelligence Journal*, 90(1-2):225–279. 29, 70

[Bonet et al., 2008] Bonet, B., Haslum, P., Hickmott, S. L., and Thiébaux, S. (2008). Directed unfolding of petri nets. *T. Petri Nets and Other Models of Concurrency*, 1:172–198. 77

[Boudol and Castellani, 1989] Boudol, G. and Castellani, I. (1989). Permutation of Transitions: An Event Structure Semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 411–427. 27, 28

[Boudol et al., 2008] Boudol, G., Castellani, I., Hennessy, M., Nielsen, M., and Winskel, G. (2008). Twenty Years on: Reflections on the CEDISYS Project. Combining True Concurrency with Process Algebra. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *LNCS*, pages 757–777. 27

[Boutrous-Saab et al., 2009] Boutrous-Saab, C., Coulibaly, D., Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. (2009). An integrated framework for web services orchestration. *Int. J. Web Service Res.*, 6(4):1–29. 10

[Bozkurt et al., 2010] Bozkurt, M., Harman, M., and Hassoun, Y. (2010). Testing Web Services: A Survey. Technical Report TR-10-01, Centre for Research on Evolution, Search & Testing, King's College London. 59

[Bracciali et al., 2005] Bracciali, A., Brogi, A., and Canal, C. (2005). A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54. 6, 33, 35

[Bravetti et al., 2011] Bravetti, M., Giusto, C. D., Pérez, J. A., and Zavattaro, G. (2011). Adaptable processes (extended abstract). In *FMOODS/FORTE*, pages 90–105. 35

[Brogi et al., 2004] Brogi, A., Canal, C., and Pimentel, E. (2004). Behavioural Types and Component Adaptation. In *Proc. of AMAST'04*, volume 3116 of *LNCS*, pages 42–56. Springer. 35

[Brogi et al., 2006a] Brogi, A., Canal, C., and Pimentel, E. (2006a). Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 63(1):39–56. 33

[Brogi and Corfini, 2007] Brogi, A. and Corfini, S. (2007). Behaviour-aware discovery of web service compositions. *Int. J. Web Service Res.*, 4(3):1–25. 14

[Brogi et al., 2006b] Brogi, A., Corfini, S., Aldana, J. F., and Navas, I. (2006b). Automated Discovery of Compositions of Services Described with Separate Ontologies. In *Proc. of ICSOC'06*. 34, 50

[Brogi and Popescu, 2005] Brogi, A. and Popescu, R. (2005). Towards Semi-automated Workflow-based Aggregation of Web Services. In *Proc. of ICSOC*. 37

[Brogi and Popescu, 2006] Brogi, A. and Popescu, R. (2006). Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294, pages 27–39. 33

[Brogi and Popescu, 2007] Brogi, A. and Popescu, R. (2007). Service Adaptation through Trace Inspection. *Int. Journal of Business Process Integration and Management*, 2(1):9–16. 34

[Bucchiarone et al., 2007] Bucchiarone, A., Melgratti, H., and Severoni, F. (2007). Testing service composition. In *8th Argentine Symposium on Software Engineering (ASSE'07)*. 59

[Bultan and Fu, 2008] Bultan, T. and Fu, X. (2008). Specification of Realizable Service Conversations using Collaboration Diagrams. *Service Oriented Computing and Applications*, 2(1):27–39. 61

[Busi et al., 2006] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. (2006). Choreography and orchestration conformance for system design. In *Proc. of Coordination'06*. 12, 60, 61

[Cabot et al., 2007] Cabot, J., Clarisó, R., and Riera, D. (2007). UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In *Proc. of ASE*. 65

[Cámara et al., 2009] Cámara, J., Martín, J. A., Salaün, G., Cubo, J., Ouederni, M., Canal, C., and Pimentel, E. (2009). ITACA: An integrated toolbox for the automatic composition and adaptation of Web services. In *ICSE*, pages 627–630. 36

[Cámara et al., 2008] Cámara, J., Salaün, G., and Canal, C. (2008). Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *Journal of Universal Computer Science*, 14(13):2182–2211. 35

[Cámara et al., 2009] Cámara, J., Salaün, G., Canal, C., and Ouederni, M. (2009). Interactive Specification and Verification of Behavioural Adaptation Contracts. In *Proc. of QSIC'09*, pages 65–75. IEEE Computer Society. 44

[Canal et al., 2006] Canal, C., Murillo, J. M., and Poizat, P. (2006). Software Adaptation. *L'Objet*, 12(1):9–31. 5, 33

[Canfora and Penta, 2009] Canfora, G. and Penta, M. D. (2009). Service-oriented architectures testing: A survey. In *ISSSE'08*, pages 78–105. 59

[Carbone et al., 2007] Carbone, M., Honda, K., and Yoshida, N. (2007). Structured Communication-Centred Programming for Web Services. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 2–17. 61

[Cavallaro et al., 2009] Cavallaro, L., Nitto, E. D., and Pradella, M. (2009). An automatic approach to enable replacement of conversational services. In *Proc. of ICSOC/ServiceWave*, pages 159–174. 34

[Chafle et al., 2005] Chafle, G., Chandra, S., Mann, V., and Gowri Nanda, M. (2005). Orchestrating Composite Web Services Under Data Flow Constraints. In *Proc. of ICWS'05*, pages 211–218. IEEE Computer Society Press. 34, 61

[Champelovier et al., 2010] Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Powazny, V., Lang, F., Serwe, W., and Smeding, G. (2010). Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.1). INRIA/VASY, 109 pages. 68

[Chan et al., 2007] Chan, K. S. M., Bishop, J., and Baresi, L. (2007). Survey and comparison of planning techniques for web service composition. Technical report, Dept Computer Science, University of Pretoria. 37

[Chanthery and Pencolé, 2009] Chanthery, E. and Pencolé, Y. (2009). Principles of self-maintenance in an on-board architecture including active diagnosis. In *Proc. of the IJCAI-09 Workshop on Self-* and Autonomous Systems: reasoning and integration challenges (SAS'09)*. 77

[Cheng et al., 2008] Cheng, B. H. C., Giese, H., Inverardi, P., Magee, J., and de Lemos, R. (2008). Software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. 20

[Combemale et al., 2009] Combemale, B., Crégut, X., Garoche, P.-L., and Thirioux, X. (2009). Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *JSW*, 4(9):943–958. 21

[Constantinescu et al., 2006] Constantinescu, I., Binder, W., and Faltings, B. (2006). Service Composition with Directories. In *Proc. of SC*. 37

[Cubo et al., 2007a] Cubo, J., Salaün, G., Cámara, J., Canal, C., and Pimentel, E. (2007a). Context-based adaptation of component behavioural interfaces. In *COORDINATION*, pages 305–323. 35

[Cubo et al., 2007b] Cubo, J., Salaün, G., Canal, C., Pimentel, E., and Poizat, P. (2007b). A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, volume 215 of *ENTCS*, pages 39–55. Elsevier. 48

[Dahl and Nygaard, 1966] Dahl, O.-J. and Nygaard, K. (1966). SIMULA — an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678. 2

[de Moura and Björner, 2008] de Moura, L. M. and Björner, N. (2008). Z3: An Efficient SMT Solver. In *Proc. of TACAS'08*, pages 337–340. 65

[Decker et al., 2008] Decker, G., Kopp, O., and Barros, A. (2008). An Introduction to Service Choreographies. *Information Technology*, 50(2):122–127. 12

[DeRemer and Kron, 1976] DeRemer, F. and Kron, H. H. (1976). Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Trans. Software Eng.*, 2(2):80–86. 2

[Dumas et al., 2006] Dumas, M., Wang, K. W. S., and Spork, M. L. (2006). Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proc. of BPM'06*, volume 4102, pages 65–80. 34

[Dustdar and Schreiner, 2005] Dustdar, S. and Schreiner, W. (2005). A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30. 37

[El Haddad et al., 2010] El Haddad, J., Manouvrier, M., and Rukoz, M. (2010). TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition. *IEEE T. Services Computing*, 3(1):73–85. 7, 14

[El Haddad et al., 2009] El Haddad, J., Melliti, T., and Poizat, P. (2009). State of the art: languages for services interface description and for services composition. `http://anr-perso.ibisc.univ-evry.fr/PublicDocuments/`. 8, 10

[Endo and Simao, 2010] Endo, A. T. and Simao, A. S. (2010). A systematic review on formal testing approaches for web services. In *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST 2010)*, pages 89–98. 59

[Esparza et al., 2010] Esparza, J., Leucker, M., and Schlund, M. (2010). Learning workflow petri nets. In *Petri Nets*, pages 206–225. 77

[Esparza et al., 2002] Esparza, J., Römer, S., and Vogler, W. (2002). An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310. 77

[Fiadeiro et al., 2006] Fiadeiro, J., Lopes, A., and Bocchi, L. (2006). A Formal Approach to Service Component Architecture. *Proceedings of Web Services and Formal Methods (WSFM'07)*, 4184:193–213. 11

[Filman et al., 2005] Filman, R. E., Elrad, T., Clarke, S., and Akşit, M., editors (2005). *Aspect-Oriented Software Development*. Addison-Wesley. 2

[Foster et al., 2010] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2010). An Integrated Workbench for Model-Based Engineering of Service Compositions. *IEEE T. Services Computing*, 3(2):131–144. 12, 60

[Frantzen et al., 2009] Frantzen, L., Huerta, M., Kiss, Z., and Wallet, T. (2009). On-The-Fly Model-Based Testing of Web Services with Jambition. In *Proc. of WS-FM*, volume 5387 of *LNCS*. 60

[Frantzen et al., 2006] Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2006). A Symbolic Framework for Model-Based Testing. In *Proc. of FATES/RV*, volume 4262 of *LNCS*. 60

[Fu et al., 2004] Fu, X., Bultan, T., and Su, J. (2004). Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2):19–37. 61

[Gaston et al., 2006] Gaston, C., Le Gall, P., Rapin, N., and Touil, A. (2006). Symbolic Execution Techniques for Test Purpose Definition. In *Proc. of TESTCOM*, volume 3964 of *LNCS*. 60

[Gaudel, 2011] Gaudel, M.-C. (2011). Checking models, proving programs, and testing systems. In *TAP'2011*, pages 1–13. 13

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers. 29

[Gierds et al., 2010] Gierds, C., Mooij, A. J., and Wolf, K. (2010). Reducing adapter synthesis to controller synthesis. *IEEE Transactions on Services Computing*, 99(PrePrints). 34, 37

[Gowri Nanda et al., 2004] Gowri Nanda, M., Chandra, S., and Sarkar, V. (2004). Decentralizing Execution of Composite Web Services. In *Proc. of OOPSLA'04*, pages 170–187. ACM Computer Society Press. 34, 36, 61

[Grigori et al., 2008] Grigori, D., Corrales, J. C., and Bouzeghoub, M. (2008). Behavioral matchmaking for service retrieval: Application to conversation protocols. *Inf. Syst.*, 33(7-8):681–698. 6, 14, 35

[Hacid et al., 2009] Hacid, M.-S., Lécué, F., Léger, A., Rey, C., and Toumani, F. (2009). Les web services sémantiques, automate et intégration i. introduction, éléments et scénarios, découverte de services web. *Technique et Science Informatiques*, 28(2):229–262. 6, 14

[Haddad et al., 2004a] Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. (2004a). A dense time semantics for Web services specifications languages. In *Proc. of ICTTA'04*. 24

[Haddad et al., 2004b] Haddad, S., Melliti, T., Moreaux, P., and Rampacek, S. (2004b). Modelling Web Services Interoperability. In *Proc. of ICEIS'04*. 36, 51

[Hayman and Winskel, 2008] Hayman, J. and Winskel, G. (2008). The unfolding of general petri nets. In *FSTTCS'08*, pages 223–234. 77

[Hemer, 2005] Hemer, D. (2005). A formal approach to component adaptation and composition. In *ACSC*, pages 259–266. 35, 77

[Hennessy and Lin, 1995] Hennessy, M. and Lin, H. (1995). Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389. 26

[Hickmott et al., 2007] Hickmott, S. L., Rintanen, J., Thiébaux, S., and White, L. B. (2007). Planning via petri net unfolding. In *IJCAI*, pages 1904–1911. 77

[Hopcroft and Ullman, 1979] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. 40

[Inverardi et al., 2005] Inverardi, P., Mostarda, L., Tivoli, M., and Autili, M. (2005). Synthesis of Correct and Distributed Adaptors for Component-based Systems: an Automatic Approach. In *Proc. of ASE'05*, pages 405–409. ACM Press. 34, 36, 61

[Inverardi et al., 2011] Inverardi, P., Spalazzese, R., and Tivoli, M. (2011). Application-layer connector synthesis. In *SFM*, pages 148–190. 33, 34, 35

[Inverardi and Tivoli, 2003] Inverardi, P. and Tivoli, M. (2003). Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183. 6, 33, 34, 35, 36, 37, 41, 49

[Inverardi and Tivoli, 2007] Inverardi, P. and Tivoli, M. (2007). A reuse-based approach to the correct and automatic composition of web-services. In *ESSPE*, pages 29–33. 34, 37

[ISO/IEC, 1989] ISO/IEC (1989). LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO. 45

[Jeannet et al., 2005] Jeannet, B., Jéron, T., Rusu, V., and Zinovieva, E. (2005). Symbolic Test Selection based on Approximate Analysis. In *Proc. of TACAS*, volume 3440 of *LNCS*. 60

[Jensen et al., 2007] Jensen, K., Kristensen, L. M., and Wells, L. (2007). Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *STTT*, 9(3-4):213–254. 77

[Jonsson, 2011] Jonsson, B. (2011). Learning of automata models extended with data. In *SFM'11*, pages 327–349. 77

[Kazhamiakin and Pistore, 2006] Kazhamiakin, R. and Pistore, M. (2006). Analysis of Realizability Conditions for Web Service Choreographies. In *Proc. of FORTE'06*, pages 61–76. 76

[Khurshid et al., 2003] Khurshid, S., Pasareanu, C. S., and Visser, W. (2003). Generalized Symbolic Execution for Model Checking and Testing. In *Proc. of TACAS*, volume 2619 of *LNCS*. 27

[Kiepuszewski, 2003] Kiepuszewski, B. (2003). *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflow*. PhD thesis, Queensland University of Technology, Brisbane, Australia. 11, 55

[Kindler and Petrucci, 2009] Kindler, E. and Petrucci, L. (2009). Towards a Standard for Modular Petri Nets: A Formalisation. In *Proc. of Applications and Theory of Petri Nets (ATPN'09)*, pages 43–62. 29

[King, 1976] King, J. C. (1976). Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394. 27

[Klush et al., 2005] Klush, M., Gerber, A., and Schmidt, M. (2005). Semantic Web Service Composition Planning with OWLS-Xplan. In *Proc. of the AAAI Fall Symposium on Agents and the Semantic Web*. 37

[Kordon et al., 2008] Kordon, F., Hugues, J., and Renault, X. (2008). From model driven engineering to verification driven engineering. In *SEUS*, pages 381–393. 21, 22

[Kozen, 1983] Kozen, D. (1983). Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354. 47

[Li et al., 2007] Li, J., Zhu, H., and Pu, G. (2007). Conformance Validation between Choreography and Orchestration. In *Proc. of TASE'07*. 14, 61

[Liu et al., 2007] Liu, Z., Ranganathan, A., and Riabov, A. (2007). Modeling Web Services using Semantic Graph Transformation to Aid Automatic Composition. In *Proc. of ICWS*. 37

[Lohmann and Wolf, 2010] Lohmann, N. and Wolf, K. (2010). Realizability Is Controllability. In *Proc. of WS-FM'09*, volume 6194 of *LNCS*, pages 110–127. Springer. 61

[Marconi and Pistore, 2009] Marconi, A. and Pistore, M. (2009). Synthesis and Composition of Web Services. In *Proc. of the 9th International School on Formal Methods for the Design of Computer, Communications and Software Systems: Web Services (SFM'09)*, pages 89–157. 15, 37, 48, 62

[Martín and Pimentel, 2009] Martín, J. A. and Pimentel, E. (2009). Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, volume 229 of *ENTCS*, pages 115–131. 44

[Mateescu, 2006] Mateescu, R. (2006). Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 8(1):37–56. Full version available as INRIA Research Report RR-5948, July 2006. 47

[Mateescu and Rampacek, 2008] Mateescu, R. and Rampacek, S. (2008). Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In *Advances in Enterprise Engineering I*, volume 10, pages 179–193. 63

[Mayer et al., 2008] Mayer, P., Schroeder, A., and Koch, N. (2008). A Model-Driven Approach to Service Orchestration. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008)*, volume 2, pages 533–536. IEEE Computer Society. 11

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. R. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. 26(1):70–93. 3

[Mens and Demeyer, 2008] Mens, T. and Demeyer, S., editors (2008). *Software Evolution*. Springer. 17

[Meyer, 1997] Meyer, B. (1997). *Object-oriented software construction*. Prentice Hall. 2

[Morel and Alexander, 2004] Morel, B. and Alexander, P. (2004). Spartacas automating component reuse and adaptation. *IEEE Trans. Software Eng.*, 30(9):587–600. 35, 77

[Moser et al., 2008] Moser, O., Rosenberg, F., and Dustdar, S. (2008). Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proc. of WWW*, pages 815–824. 35

[Motahari-Nezhad et al., 2007] Motahari-Nezhad, H. R., Benatallah, B., Martens, A., Curbera, F., and Casati, F. (2007). Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, pages 993–1002. 34

[Motahari-Nezhad et al., 2010] Motahari-Nezhad, H. R., Xu, G. Y., and Benatallah, B. (2010). Protocol-aware matching of web service interfaces for adapter development. In *Proc. of WWW'10*, pages 731–740. 34

[Musaraj et al., 2010] Musaraj, K., Yoshida, T., Daniel, F., Hacid, M.-S., Casati, F., and Benatallah, B. (2010). Message Correlation and Web Service Protocol Mining from Inaccurate Logs. In *Proc. of ICWS'10*, pages 259–266. 6, 77

[OASIS, 2007] OASIS (2007). *Web Services Business Process Execution Language Version 2.0*. OASIS. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 10, 11

[OMG, 2005] OMG (2005). *Unified Modeling Language: Superstructure – Version 2.0*. OMG. http://www.omg.org/spec/UML/2.0/. 11

[OMG, 2011] OMG (2011). *Business Process Model and Notation (BPMN) – Version 2.0*. OMG. http://www.omg.org/spec/BPMN/2.0/. 11, 61, 69, 76

[Open SOA, 2007] Open SOA (2007). Service Component Architecture Home. http://osoa.org/display/Main/Service+Component+Architecture+Home. 7

[Ouederni et al., 2011] Ouederni, M., Salaün, G., and Pimentel, E. (2011). Measuring the compatibility of service interaction protocols. In *SAC'11*, pages 1560–1567. 35

[Oxford University Press, 2010] Oxford University Press (2010). Oxford Dictionaries Online. http://oxforddictionaries.com/. 4

[Padovani, 2009] Padovani, L. (2009). Contract-Based Discovery and Adaptation of Web Services. In *Proc. of the 9th International School on Formal Methods for the Design of Computer, Communications and Software Systems: Web Services (SFM'09)*, volume 5569 of *LNCS*, pages 213–260. Springer. 35

[Papadopoulos and Arbab, 1998] Papadopoulos, G. A. and Arbab, F. (1998). Coordination models and languages. *Advances in Computers*, 46:330–401. 3

[Papazoglou and Georgakopoulos, 2003] Papazoglou, M. P. and Georgakopoulos, D. (2003). Introduction. *Communications of the ACM*, 46(10):24–28. Introduction to the special issue on Service-Oriented Computing. 2

[Papazoglou and van den Heuvel, 2007] Papazoglou, M. P. and van den Heuvel, W.-J. (2007). Service oriented architectures: approaches, technologies and research issues. *VLDB J.*, 16(3):389–415. 20

[Parnas, 1972] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:1053–1058. 2

[Peer, 2005] Peer, J. (2005). Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen. 37

[Pelliccione et al., 2008] Pelliccione, P., Tivoli, M., Bucchiarone, A., and Polini, A. (2008). An architectural approach to the correct and automatic assembly of evolving component-based systems. *Journal of Systems and Software*, 81(12):2237–2251. 35

[Plášil and Visnovsky, 2002] Plášil, F. and Visnovsky, S. (2002). Behavior Protocols for Software Components. 28(11):1056–1076. 6, 7

[Pohl et al., 2005] Pohl, K., Böckle, G., and van der Linden, F. J., editors (2005). *Software Product Line Engineering. Foundations, Principles and Techniques.* Springer. 2, 17

[Poizat and Royer, 2006] Poizat, P. and Royer, J.-C. (2006). A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. *Journal of Universal Computer Science*, 12(12):1741–1782. 6, 26

[Qiu et al., 2007] Qiu, Z., Zhao, X., Cai, C., and Yang, H. (2007). Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*. 12, 14, 60, 61, 66

[Ramadge and Wonham, 1989] Ramadge, P. J. G. and Wonham, W. M. (1989). The Control of Discrete Event Systems. *Proc. of the IEEE*, 77(1):81–98. 17

[Rao and Su, 2004] Rao, J. and Su, X. (2004). A Survey of Automated Web Service Composition Methods. In *Proc. of SWSWPC*. 37

[Reisig, 1985] Reisig, W. (1985). *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer. 28, 29

[Reisig, 2009] Reisig, W. (2009). Simple Composition of Nets. In *Proc. of Applications and Theory of Petri Nets (ATPN'09)*, pages 23–42. 29

[Reussner, 2003] Reussner, R. H. (2003). Automatic Component Protocol Adaptation with the CoConut/J Tool Suite. *Future Generation Computer Systems*, 19(1):627–639. 33

[Roohi and Salaün, 2011] Roohi, N. and Salaün, G. (2011). Realizability and Dynamic Reconfiguration of Chor Specifications. *Informatica*, 35(1):39–49. 12, 60

[Rusli et al., 2011] Rusli, H. M., Ibrahim, S., Puteh, M., and Tabatabaei, S. G. H. (2011). A comparative evaluation of state-of-the-art web service composition testing approaches. In *6th IEEE/ACM International Workshop on Automation of Software Test (AST 2011), workshop at ICSE 2011*. 59, 60

[Salaün and Bultan, 2009] Salaün, G. and Bultan, T. (2009). Realizability of Choreographies using Process Algebra Encodings. In *Proc. of IFM'09*. 61

[Schmidt and Reussner, 2002] Schmidt, H. W. and Reussner, R. H. (2002). Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS'02*, pages 213–229. 34

[Seguel et al., 2008] Seguel, R., Eshuis, R., and Grefen, P. (2008). An Overview on Protocol Adaptors for Service Component Integration. BETA Working Paper Series WP 265, Eindhoven University of Technology. 17, 33, 62

[Seguel et al., 2010] Seguel, R., Eshuis, R., and Grefen, P. (2010). Generating minimal protocol adaptors for loosely coupled services. In *International Conference on Web Services*, pages 417–424. 34

[Shan et al., 2010] Shan, Z., Kumar, A., and Grefen, P. W. P. J. (2010). Towards integrated service adaptation. a new approach combining message and control flow adaptation. In *ICWS*, pages 385–392. 34

[Sinha and Paradkar, 2006] Sinha, A. and Paradkar, A. M. (2006). Model-based functional conformance testing of web services operating on persistent data. In *TAV-WEB*, pages 17–22. 60

[Steffen et al., 2011] Steffen, B., Howar, F., and Merten, M. (2011). Introduction to active automata learning from a practical perspective. In *SFM'11*, pages 256–296. 6, 77

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley. 2, 3

[Taher et al., 2009] Taher, Y., Aït-Bachir, A., Fauvet, M.-C., and Benslimane, D. (2009). Diagnosing Incompatibilities in Web Service Interactions for Automatic Generation of Adapters. In *Proc. of AINA'09*, pages 652–659. IEEE Computer Society. 35

[ter Beek et al., 2007] ter Beek, M., Bucchiarone, A., and Gnesi, S. (2007). Formal Methods for Service Composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10. 12, 63

[Tivoli et al., 2007] Tivoli, M., Fradet, P., Girault, A., and Gößler, G. (2007). Adaptor synthesis for real-time components. In *TACAS*, pages 185–200. 35

[Tivoli and Inverardi, 2008] Tivoli, M. and Inverardi, P. (2008). Failure-free coordinators synthesis for component-based architectures. *Science of Computer Programming*, 71(3):181–212. 34

[Toörmä et al., 2008] Toörmä, S., Villstedt, J., Lehtinen, V., Oliver, I., and Luukkala, V. (2008). Semantic Web Services – A Survey. Technical Report TKK-TKO-B158, Helsinki University of Technology, Department of Computer Science and Engineering. 6

[Uchitel et al., 2003] Uchitel, S., Kramer, J., and Magee, J. (2003). Synthesis of Behavioural Models from Scenarios. 29(2):99–115. 40, 64

[Uchitel et al., 2004] Uchitel, S., Kramer, J., and Magee, J. (2004). Incremental Elaboration of Scenario-based Specifications and Behavior Models using Implied Scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85. 61

[Vallecillo et al., 2000] Vallecillo, A., Hernández, J., and Troya, J. M. (2000). New issues in object interoperability. In *Object-Oriented Technology*, volume 1964, pages 256–269. 6

[Vallecillo et al., 2003] Vallecillo, A., Vasconcelos, V. T., and Ravara, A. (2003). Typing the Behavior of Objects and Component Using Session Types. *Electr. Notes Theor. Comput. Sci.*, 68(3). 35

[van der Aalst, 2005] van der Aalst, W. M. P. (2005). Pi Calculus Versus Petri Nets. Let us eat "humble pie" rather than further inflate the "Pi hype". *BPTrends*, 3(5):1–11. 29

[van der Aalst, 2011] van der Aalst, W. M. P. (2011). *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer. 6, 77

[van der Aalst et al., 2009] van der Aalst, W. M. P., Mooij, A. J., Stahl, C., and Wolf, K. (2009). Service Interaction: Patterns, Formalization, and Analysis. In *Proc. of Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'2009)*, volume 5569 of *Lecture Notes in Computer Science*, pages 42–88. 34, 37

[van der Aalst et al., 2003] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51. 11

[van Glabbeek, 2001] van Glabbeek, R. J. (2001). *Handbook of Process Algebra*, chapter The linear time - branching time spectrum I. Elsevier, North Holland. 25

[Vardi, 2001] Vardi, M. Y. (2001). Branching vs. linear time: Final showdown. In *Proc. of TACAS'01*, pages 1–22. 24

[W3C, 2001] W3C (2001). *Web Services Description Language (WSDL) 1.1*. W3C. http://www.w3.org/TR/wsdl. 5, 8

[W3C, 2004] W3C (2004). *Web Service Choreography Description Language (WS-CDL) 1.0*. W3C. http://www.w3.org/TR/ws-cdl-10/. 66

[W3C, 2007] W3C (2007). Semantic annotations for wsdl and xml schema (sawsdl). http://www.w3.org/TR/sawsdl/. 50, 71

[Wang et al., 2008] Wang, K., Dumas, M., Ouyang, C., and Vayssière, J. (2008). The Service Adaptation Machine. In *Proc. of ECOWS'08*, pages 145–154. IEEE Computer Society. 35

[Winskel, 1986] Winskel, G. (1986). Event Structures. In *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. 27, 29

[Wonham and Ramadge, 1987] Wonham, W. M. and Ramadge, P. J. (1987). On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, 25(3):637–659. 17

[Xiong et al., 2010] Xiong, P., Fan, Y., and Zhou, M. (2010). A Petri Net Approach to Analysis and Composition of Web Services. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 40(2):376–387. 34, 37

[Yan et al., 2006] Yan, J., Li, Z. J., Yuan, Y., Sun, W., and Zhang, J. (2006). Bpel4ws unit testing: Test case generation using a concurrent path analysis approach. In *ISSRE*, pages 75–84. 60

[Yellin and Strom, 1997] Yellin, D. M. and Strom, R. E. (1997). Protocol Specifications and Components Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333. 33

[Zeng et al., 2004] Zeng, L., Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., and Chang, H. (2004). Qos-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327. 7, 14

[Zheng and Yan, 2008] Zheng, X. and Yan, Y. (2008). An efficient syntactic web service composition algorithm based on the planning graph model. In *ICWS*, pages 691–699. 37

[Zheng et al., 2007] Zheng, Y., Zhou, J., and Krause, P. (2007). An Automatic Test Case Generation Framework for Web Services. *Journal of Software*, 2(3):64–77. 60

[Zhou et al., 2010] Zhou, L., Ping, J., Xiao, H., Wang, Z., Pu, G., and Ding, Z. (2010). Automatically Testing Web Services Choreography with Assertions. In *Proc. of ICFEM'10*. 60