On Behavioural Interfaces and Contracts for Software Adaptation

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

Department of Computer Science, University of Málaga, Spain

Abstract

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

1. Introduction

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

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

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

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

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

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

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

2. Behavioural Interfaces and Adaptation Contracts

2.1. Behavioural Interfaces

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

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

2.2. Adaptation Contract Specification

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

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

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

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

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

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

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

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

3. Adaptor Generation and Implementation

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

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

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

4. Tool Support

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

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

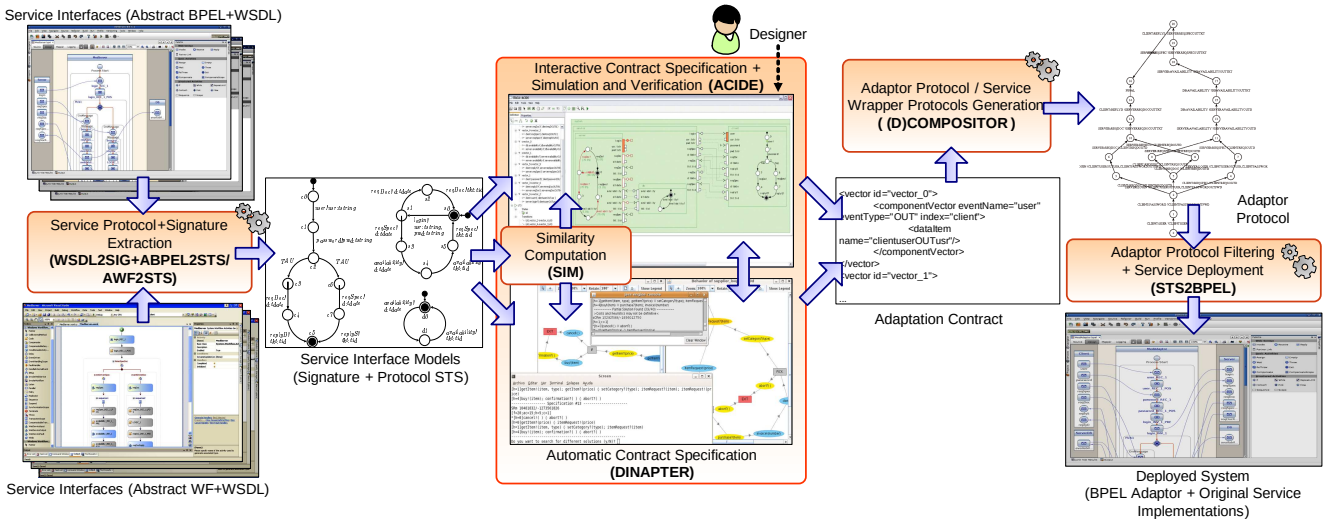

Figure 1. Adaptation process overview in ITACA

5. Concluding Remarks

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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