Adaptation of Asynchronously Communicating Software

Carlos Canal[1] and Gwen Salaün[2]

[1] University of Malaga, Spain
[2] University of Grenoble Alpes, Inria, LIG, CNRS, France

Abstract. Software adaptation techniques aim at generating new components called adapters, which make a set of services work correctly together by compensating for existing mismatch. Most approaches assume that services interact synchronously using rendez-vous communication. In this paper, we focus on asynchronous communication, where services interact exchanging messages via buffers. We overview a method for automatically generating adapters in such asynchronous environments.

1 Introduction

Software Adaptation [22, 7] is a non-intrusive solution for composing black-box software services (*peers* in this paper) whose functionality is as required for the new system, but that present interface mismatch which leads to deadlock or other undesirable behaviour when peers are combined. Adaptation techniques aim at automatically generating new components called *adapters*, and usually rely on an *adaptation contract*, which is an abstract description of how mismatch can be worked out. All interactions pass through the adapter, which acts as an orchestrator and makes the involved peers work correctly together by compensating for mismatch. Many solutions have been proposed since the seminal work by Yellin and Strom [22], see, *e.g.*, [3, 5, 20, 15, 12, 13]. Most existing approaches assume that peers interact using synchronous communication, that is rendez-vous synchronizations. Nonetheless, asynchronous communication, *i.e.*, communication via buffers, is now omnipresent in areas such as cloud computing or Web development. Asynchronous communication complicates the adapter generation process, because the corresponding systems are not necessarily bounded and may result into infinite systems [4].

In this paper, we rely on the *synchronizability* property [1, 17] in order to propose an approach for generating adapters for peers interacting asynchronously via (possibly unbounded) FIFO buffers. A set of peers is synchronizable if and only if the system generates the same sequences of messages under synchronous and unbounded asynchronous communication, considering only the ordering of the send actions and ignoring the ordering of receive actions. Synchronizability can be verified by checking the equivalence of the synchronous version of a given system with its 1-bounded asynchronous version (in which each peer is equipped with one input FIFO buffer bounded to size 1). Thus, this property can be analysed using equivalence checking techniques on finite systems.

More precisely, given a set of peers modelled using Labelled Transition Systems and an adaptation contract, we first reuse existing adapter generation techniques for synchronous communication, *e.g.*, [8, 15]. Then, we consider a system composed of a set of peers interacting through the generated adapter, and we check whether that system satisfies the synchronizability property. If this is the case, this means that the system will behave exactly the same whatever bound we choose for buffers, therefore this adapter is a solution to our composition problem. If synchronizability is not preserved, a counterexample is returned, which can be used for refining the adaptation contract, until preserving synchronizability. It is worth observing that the main reason for non-synchronizability is due to emissions, which are uncontrollable in an asynchronous environment, hence have to be considered properly in the adaptation contract.

The organization of this paper is as follows. Section 2 defines our models for peers and introduces the basics on synchronous software adaptation. Section 3 presents our results on the generation of adapters assuming that peers interact asynchronously. Finally, Section 4 reviews related work and Section 5 concludes.

2 Synchronous Adaptation

We assume that peers are described using a behavioural interface in the form of a Labelled Transition System. A Labelled Transition System (LTS) is a tuple (S, s^0, Σ, T) where S is a set of states, $s^0 \in S$ is the initial state, $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$ is a finite alphabet partitioned into a set $\Sigma^!$ ($\Sigma^?$, resp.) of send (receive, resp.) messages and the internal action τ, and $T \subseteq S \times \Sigma \times S$ is the transition relation.

The alphabet of the LTS is built on the set of operations used by the peer in its interaction with the world. This means that for each operation p provided by the peer, there is an event $p? \in \Sigma^?$ in the alphabet, and for each operation r required from its environment, there is an event $r! \in \Sigma^!$. Events with the same name and opposite directions ($a!$, $a?$) are complementary, and their match stands for inter-peer communication through message-passing. Additionally to peer communication events, we assume that the alphabet also contains a special τ event to denote internal (not communicating) behaviour. Note that as usually done in the literature [14, 10, 19], our interfaces abstract from operation arguments, types of return values, and exceptions. Nevertheless, they can be easily extended to explicitly represent operation arguments and their associated data types, by using Symbolic Transition Systems (STSs) [15] instead of LTSs.

Example 1. We use as running example an online hardware supplier. This example was originally presented in [9] and both participants (a supplier and a buyer) were implemented using the Microsoft WF/.NET technology. Figure 1 presents the LTSs corresponding to both peers. The supplier first receives a request under the form of two messages that indicate the reference of the requested hardware (`type`), and the max price to pay (`price`). Then, it sends a response indicating if the request can be replied positively or not (`reply`). Next, the supplier may receive and reply other requests, or receive an order of purchase on the last

reference requested (`buy`). In the latter case, a confirmation is sent (`ack`). The behaviour of the buyer starts by submitting a request (`request`). Upon reception of the response (`reply`), the buyer either submits another request, buys the requested product (`purchase` and `ack`), or ends the session (`stop`).

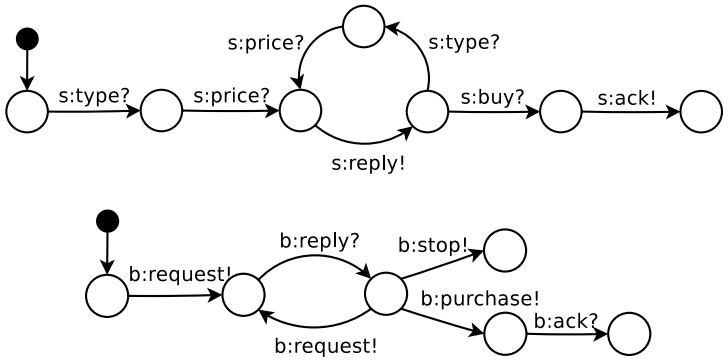

Fig. 1. LTS interfaces of supplier (top) and buyer (bottom) peers

As shown in the example, typical mismatch situations appear when event names do not correspond, the order of events is not respected, or an event in one peer has no counterpart or matches several events in another one. All these cases of behavioural mismatch can be worked out by specifying adaptation rules. Adaptation rules express correspondences between operations of the peers, like bindings between ports or connectors in architectural descriptions. Rules are given as adaptation vectors. An adaptation vector (or vector for short) for a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ with $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$, is a tuple $\langle e_1, \ldots, e_n \rangle$ with $e_i \in \Sigma_i \cup \{\epsilon\}$, ϵ meaning that a peer does not participate in the interaction.

In order to unambiguously identify them, event names may be prefixed by the name of the peer, *e.g.*, $\mathcal{P}_i : p?$, or $\mathcal{P}_j : r!$, and in that case ϵ can be omitted. For instance, the vector $\langle p_1 : a!, p_2 : \epsilon, p_3 : b?, p_4 : c? \rangle$ represents an adaptation rule indicating that the output event $a!$ from peer p_1 should match both input events $b?$ and $c?$ in p_3, and p_4 respectively, while peer $p2$ does not participate in this interaction. For more details on the syntax and expressiveness of adaptation vectors, we refer to [8].

An adaptation contract for a set of peers is a set of adaptation vectors for those peers. Writing the adaptation contract is the only step of our approach which is not handled automatically. This step is crucial because an inadequate contract would induce the generation of an adapter that will not make the composition of peers to behave correctly (for instance, some expected interactions may be discarded by the adapter, in order to avoid deadlock). However, the adaptation methodology that we propose (Section 3) is iterative, which helps in writing the adaptation contract.

Example 2. Going back to our running example, we observe several differences between both interfaces. For instance, the buyer submits a single message for each request, while the supplier expects two messages; the name of the message for carrying out a purchase is not the same, etc. The vectors below are proposed for composing and adapting the whole system. The correspondence between `request!` and messages `type?` and `price?` can be achieved using two vectors, $\mathtt{V}_{\mathtt{req}}$ and $\mathtt{V}_{\mathtt{price}}$. The mismatch between `purchase!` and `buy?` can be solved by vector $\mathtt{V}_{\mathtt{buy}}$.

$\mathtt{V}_{\mathtt{req}} = \langle \mathtt{b{:}request!}, \mathtt{s{:}type?} \rangle$
$\mathtt{V}_{\mathtt{price}} = \langle \mathtt{b{:}}\varepsilon, \mathtt{s{:}price?} \rangle$
$\mathtt{V}_{\mathtt{reply}} = \langle \mathtt{b{:}reply?}, \mathtt{s{:}reply!} \rangle$
$\mathtt{V}_{\mathtt{buy}} = \langle \mathtt{b{:}purchase!}, \mathtt{s{:}buy?} \rangle$
$\mathtt{V}_{\mathtt{ack}} = \langle \mathtt{b{:}ack?}, \mathtt{s{:}ack!} \rangle$

In [8, 15] we have shown how an adapter can be automatically derived from a set of interfaces and an adaptation contract. Our approach relies on an encoding into process algebra together with on-the-fly exploration and reduction techniques. The adapter is given by an LTS which, put into a non-deadlock-free system yields it deadlock-free. All the exchanged events will pass through the adapter, which can be seen as a coordinator for the peers to be adapted. Code generation is also supported by our approach, thus BPEL adapters can be automatically synthesised from an adapter LTSs. All these steps are automated by the Itaca toolset [6]. Notice that the adaptation algorithms in [8, 15] generate *synchronous* adapters, that is, they assume a synchronous communication model for peers. In our present work we show how our previous results can be applied to asynchronous adaptation, where peers communicate asynchronously and are equipped with an input message buffer.

Example 3. Figure 2 presents the adapter LTS generated for our running example. Since the adapter is an additional peer through which all communications transit, all the messages appearing in the adapter LTS are reversed with respect to those in the peers. Note, for instance, how the adapter receives the request coming from the buyer, and splits this request into messages carrying the type and price information. This LTS also shows how the adapter interacts on different names (`purchase?` and `buy!`) to make the communication possible.

3 Asynchronous Adaptation

Our asynchronous adaptation techniques rely on the synchronizability property [1, 17]. A set of peers is synchronizable if and only if the system generates the same sequences of messages under synchronous and unbounded asynchronous communication, considering only the ordering of the send actions and ignoring the ordering of receive actions. Focusing only on send actions makes sense for verification purposes because: (i) send actions are the actions that transfer messages to the network and are therefore observable, (ii) receive actions correspond

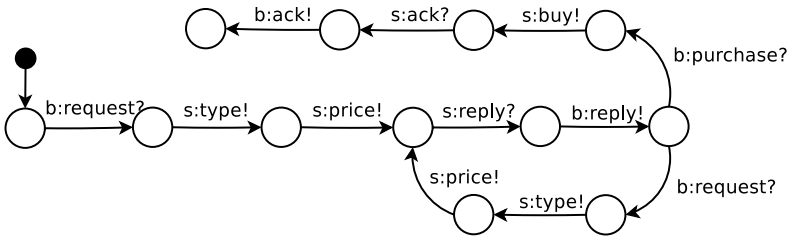

Fig. 2. Adapter LTS for the case study

to local consumptions by peers from their buffers and can therefore be considered to be local and private information. Synchronizability can be verified by checking the equivalence of the synchronous version of a given system with its 1-bounded asynchronous version (in which each peer is equipped with one input FIFO buffer bounded to size 1). Thus, this property can be verified using equivalence checking techniques on finite systems, although the set of peers interacting asynchronously can result in infinite systems.

The synchronizability results directly apply here, considering the adapter as a peer whose specificity is just that it interacts with all the other peers. It was proved that checking the equivalence between the synchronous composition and the 1-bounded asynchronous composition is a sufficient and necessary condition for branching synchronizability [17]. In the rest of this section, we show how we reuse the synchronizability property for generating adapters that work in asynchronous environments.

Given a set of mismatching peers modelled as LTSs and an adaptation contract (a set of vectors), an adapter LTS can be automatically synthesised as presented in Section 2. Then, we check whether the adapted synchronous composition and the 1-bounded adapted asynchronous composition are equivalent. If this is the case, it means that the system is synchronizable and its observable behaviour will remain the same whatever bound is chosen for buffers. Thus, the adapter generated using existing techniques for synchronous communication can be used as is in an asynchronous context. If the system is not synchronizable, the user refines the adaptation contract using the diagnostic returned by equivalence checking techniques. This counterexample indicates the additional behaviour present in the asynchronous composition and absent in the synchronous one, which invalidates synchronizability. The violation of this property has two main causes: either the adapter does not capture/handle all reachable emissions, or the adapter is too restrictive *wrt.* message orderings, *e.g.*, the adapter requires a sequence of two emissions, which cannot be ensured in the asynchronous composition because both emissions can be executed simultaneously. We apply iteratively this process until the synchronizability property is satisfied.

Our approach is supported by several tools: (i) we reuse the Itaca toolbox [6] for synthesising synchronous adapters, and (ii) we rely on process algebra en-

codings and reuse equivalence checking techniques available in the CADP verification toolbox [11] for checking synchronizability.

Example 4. As far as our running example is concerned, given the LTSs of the peers and the set of vectors presented in Section 2, we can automatically generate the corresponding adapter (Figure 2). However, if we check whether the composition of this adapter with the peers' LTSs satisfies synchronizability, the verdict is false, and we obtain the following counterexample: `b:request!`, `s:type!`, `s:price!`, `s:reply!`, `b:reply!`, and `b:stop!`, where the very last event appears in the asynchronous system but not in the synchronous one. Note that synchronizability focuses on emissions, hence the counterexample above contains only messages sent by a peer to the adapter (`b:request!`, `s:reply!`, `b:stop!`) or by the adapter to a peer (`s:type!`, `s:price!`, `b:reply!`). This violation is due to the fact that the emission of `stop` is not captured by any vector, and consequently it is inhibited in the synchronous system, while it is still possible in the asynchronous system because reachable emissions cannot be inhibited under asynchronous communication.

In order to correct this problem, we extend the adaptation contract by adding the following vector: $\mathtt{V}_{\mathtt{stop}} = \langle \mathtt{b:stop!}, \mathtt{s}:\varepsilon \rangle$. The corresponding adapter is generated and shown in Figure 3. The system composed of the two peers interacting through this adapter turns out to satisfy the synchronizability property. This means that the adapter can be used under asynchronous communication and the system will behave exactly the same whatever bound is chosen for buffers or if buffers are unbounded.

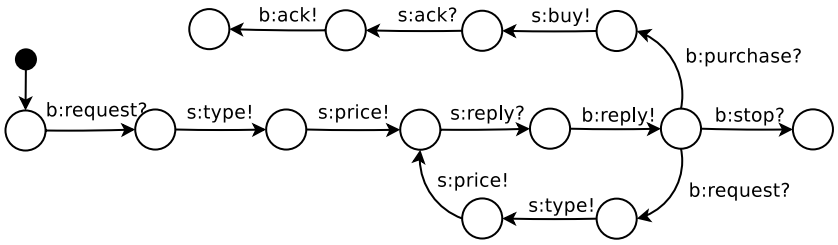

Fig. 3. Adapter LTS generated after addition of $\mathtt{V}_{\mathtt{stop}}$

4 Related Work

Existing proposals for software adaptation present interesting approaches tackling this topic from different points of view. However, most of them assume that peers interact synchronously, see, *e.g.*, [8, 21, 16, 15, 13, 2] for a few recent results. There were a few attempts to generate adapters considering asynchronous communication. Padovani [18] presents a theory based on behavioural contracts to

generate orchestrators between two services related by a subtyping (namely, subcontract) relation. This is used to generate an adapter between a client of some service S and a service replacing S. An interesting feature of this approach is its expressiveness as far as behavioural descriptions are concerned, with support for asynchronous orchestrators and infinite behaviour. The author resorts to the theory of regular trees and imposes two requirements (regularity and contractivity) on the orchestrator. However, this work does not support name mismatch nor data-related adaptation. Seguel *et al.* [20] present automatic techniques for constructing a minimal adapter for two business protocols possibly involving parallelism and loops. The approach works by assigning to loops a fixed number of iterations, whereas we do not impose any restriction, and peers may loop infinitely. Gierds and colleagues [12] present an approach for specifying behavioural adapters based on domain-specific transformation rules that reflect the elementary operations that adapters can perform. The authors also present a novel way to synthesise complex adapters that adhere to these rules by consistently separating data and control, and by using existing controller synthesis algorithms. Asynchronous adaptation is supported in this work, but buffers/places must be arbitrarily bounded for ensuring computability of the adapter.

5 Conclusion

Most existing approaches for adapting stateful software focus on systems relying on synchronous communication. In this paper, we tackle the adapter generation question from a different angle by assuming that peers interact asynchronously via FIFO buffers. This complicates the synthesis process because we may have to face infinite systems when generating the adapter behaviour. Our approach uses jointly adapter generation techniques for synchronous communication and the synchronizability property for solving this issue. This enables us to propose an iterative approach for synthesising adapters in asynchronous environments. We have applied it in this paper on a real-world example for illustration purposes.

Acknowledgements. This work was partially funded by the European Commission FP7 project SeaClouds (FP7-ICT-2013-10) and by the Spanish Government under Project TIN2012-35669.

References

1. S. Basu, T. Bultan, and M. Ouederni. Deciding Choreography Realizability. In *Proc. of POPL'12*, pages 191–202. ACM, 2012.
2. A. Bennaceur, C. Chilton, M. Isberner, and B. Jonsson. Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning. In *Proc. of SEFM'13*, volume 8137 of *LNCS*, pages 274–288. Springer, 2013.
3. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
4. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.

5. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer-Verlag, 2006.
6. J. Cámara, J. A. Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE'09*, pages 627–630. IEEE, 2009.
7. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006.
8. C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Trans. on Software Engineering*, 34(4):546–563, 2008.
9. J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. 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, 2007.
10. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.
11. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.
12. C. Gierds, A. J. Mooij, and K. Wolf. Reducing Adapter Synthesis to Controller Synthesis. *IEEE T. Services Computing*, 5(1):72–85, 2012.
13. P. Inverardi and M. Tivoli. Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In *Proc. of ICSE'13*, pages 3–12. IEEE / ACM, 2013.
14. J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour Analysis of Software Architectures*, pages 35–49. Kluwer Academic Publishers, 1999.
15. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. *IEEE Trans. on Software Engineering*, 38(4):755–777, 2012.
16. H. R. Motahari Nezhad, G. Y. Xu, and B. Benatallah. Protocol-Aware Matching of Web Service Interfaces for Adapter Development. In *Proc. of WWW'10*, pages 731–740. ACM, 2010.
17. M. Ouederni, G. Salaün, and T. Bultan. Compatibility Checking for Asynchronously Communicating Software. In *Proc. of FACS'13*, LNCS. Springer, 2013.
18. L. Padovani. Contract-Based Discovery and Adaptation of Web Services. In *Proc. of SFM'09*, volume 5569 of *LNCS*, pages 213–260. Springer, 2009.
19. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Trans. on Software Engineering*, 28(11):1056–1076, 2002.
20. R. Seguel, R. Eshuis, and P. W. P. J. Grefen. Generating Minimal Protocol Adaptors for Loosely Coupled Services. In *Proc. of ICWS'10*, pages 417–424. IEEE Computer Society, 2010.
21. W. M. P. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf. Service Interaction: Patterns, Formalization, and Analysis. In *Proc. of SFM'09*, volume 5569 of *LNCS*, pages 42–88. Springer, 2009.
22. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.