View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional Universidad de Málaga

Model-Based Adaptation of Software Communicating via FIFO Buffers

Carlos Canal[1] and Gwen Salaün[2]

[1] University of Málaga, Spain
canal@lcc.uma.es
[2] University of Grenoble Alpes, Inria, LIG, CNRS, France
gwen.salaun@imag.fr

Abstract. Software Adaptation is a non-intrusive solution for composing black-box components or services (peers) whose individual functionality is as required for the new system, but that present interface mismatch, which leads to deadlock or other undesirable behaviour when combined. Adaptation techniques aim at automatically generating new components called adapters. All the interactions among peers pass through the adapter, which acts as an orchestrator and makes the involved peers work correctly together by compensating for mismatch. Most of the existing solutions in this field assume that peers interact synchronously using rendezvous communication. However, many application areas rely on asynchronous communication models where peers interact exchanging messages via buffers. Generating adapters in this context becomes a difficult problem because peers may exhibit cyclic behaviour, and their composition often results in infinite systems. In this paper, we present a method for automatically generating adapters in asynchronous environments where peers interact using FIFO buffers.

1 Introduction

The construction of new software in modern environments is mostly achieved by reusing and composing existing software elements. These elements (*peers* in this paper) can correspond to a large variety of software, such as Web servers, databases, Graphical User Interfaces, Software-as-a-Service in the cloud, Web services, etc. In order to make possible the composition of such heterogeneous software pieces, all peers are equipped with public interfaces, which exhibit their provided/required services as well as any other composition constraints that must be respected to ensure the correct execution of the composition-to-be. However, although a set of peers may appear as adequate for a new software system under construction, it is likely that their interfaces present mismatch, especially if they have been independently developed by third parties. Mismatch takes different forms such as disparate operation names or unspecified message receptions, and it prevents the direct assembly of the peers.

Software Adaptation [32, 11] is a non-intrusive solution for composing black-box software peers that present interface mismatch, leading 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 [32], see, *e.g.*, [6, 8, 30, 24, 18, 19].

These approaches vary in different aspects, such as expressiveness of interface descriptions (signatures, behaviour, Quality-of-Service, semantics), abstraction level (from abstract models to programming languages), algorithmic techniques (discrete controller synthesis, planning, enumerative approaches, etc.), or application areas (software components, Web services, agent-oriented systems etc.). Most existing approaches assume that the 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 and Web development.

Asynchronous communication highly complicates the adapter generation process, because the corresponding systems are not necessarily bounded and may result into infinite systems. It is known that in this context, the verification problem and more particularly the boundedness property are undecidable for communicating finite state machines [7]. Therefore, if we want to generate an adapter in an asynchronous environment, how could we proceed? What bound should we choose for buffers during the generation process? Arbitrarily bounding buffers is an option, but we want to avoid imposing any kind of bounds on buffers, cyclic behaviour, or the number of participants, since it may unnecessarily restrict the behaviour of the whole system.

Recent results introduced the notion of *synchronizability* [3] and showed how to use it for checking certain properties on asynchronously communicating systems [26]. These results show that 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 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.

In this paper, we rely on synchronizability for generating adapters for peers interacting asynchronously via (possibly unbounded) FIFO buffers. 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.*, [13, 24]. Then, we consider the system composed of the set of peers interacting through the generated adapter, and we check whether it 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 is used for refining the adaptation contract. Our approach works iteratively by refining the 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. Our approach is supported by tools for generating the adapter (Itaca [9]) and checking synchronizability (CADP [17]), and was applied to several case studies for evaluation purposes.

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 defines the synchronizability property for adapted systems. Section 4 presents our approach for generating adapters assuming asynchronous communication semantics. Finally, Section 5 reviews related work, and Section 6 concludes. This paper builds on our previous work [14], where the proposal was only sketched.

2 Models

In this section, we first present the interface model through which peers are accessed and used. Then, we define adaptation contracts and explain briefly how adapters are generated from peer interfaces and contracts.

2.1 Interfaces

We assume that peers are described using a behavioural interface in the form of a Labelled Transition System (LTS).

Definition 1 (LTS). *A* Labelled Transition System *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 function.*

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 [20, 2, 28], 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) [24] instead of LTSs. However, this would render the definitions and results presented in this work much longer and cumbersome, without adding anything substantial to the technical aspects of our proposal.

2.2 Adaptation Contracts

Typical mismatch situations between peers 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. Adaptation rules are given as vectors, as defined below:

Definition 2 (Vector). *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 vector.*

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 $\langle p_1 : a!, p_3 : b?, p_4 : c?\rangle$ 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 vectors, we refer to [13].

In some complex adaptation scenarios, adaptation rules should be taken contextually (*i.e.*, vectors cannot be applied at any time, but only in certain situations). For this purpose, we use regular expressions (regex) on vectors, indicating a pattern for applying them that will constrain the adaptation process, enforcing additional properties on the adapter. In this work we use standard regex notation, where "|", and "$\star$" stand for alternation and Kleene star, respectively. For instance, $V_1\ (V_2 \mid V_3) \star\ V_4$ states that the vector V_1 should be applied first, followed by several uses of V_2 and V_3, and always ending with V_4. In the absence of a regex, we assume that adaptation rules are not contextually dependent and can be applied at any time during the adaptation process.

Definition 3 (Adaptation Contract). *An* adaptation contract V *for a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *is a set of adaptation vectors for those peers, together with a (possibly empty) regex describing a pattern for applying the vectors.*

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 is iterative (see Figure 3 in Section 4), which helps in writing the adaptation contract.

2.3 Adapter Generation

Given a set of interfaces and an adaptation contract, an adapter can be automatically derived using, *e.g.*, [13, 24]. This approach relies on an encoding into a 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 adapter LTSs. All these steps are automated by the Itaca toolset [9].

In order to generate an adapter, not only event correspondences stated in the adaptation rules must be taken into account, but also the LTSs describing the behaviour of the peers. Blindly applying adaptation rules without taking into account peers' behaviour may lead the system to a deadlock state if any of the events represented in a rule cannot be accepted by the corresponding peer in its current state. Hence, the adapter has not only the responsibility of following the adaptation rules and regex in the contract, but also to accommodate their use to the LTSs describing the behaviour of the peers, reordering and remembering events when required.

Notice that the adaptation algorithms in [13, 24] generate *synchronous* adapters, that is, they assume a synchronous communication model for peers. In our present work we show how these results can be applied to asynchronous adaptation, where peers communicate asynchronously and are equipped with an input message buffer.

2.4 Case Study

This section describes the case study that will be used as running example throughout this work. Let us consider a multi-cloud scenario for creating virtual machines (VM) in IaaS clouds such as Google Compute Engine or Windows Azure. These clouds offer different APIs for VM creation and management, which allows us to show how adaptation can solve mismatch. The names of the events used here are inspired by the actual names of the corresponding operations in Google and Azure, although the scenario is conveniently simplified in order to abstract from many details that would make it unnecessarily long and complex.

The core element of the system is a Deployment Manager (DM). The LTS describing its behaviour is shown in Figure 1 (d). After receiving a request (`request?`), the DM creates a new VM instance (`instantiate!`), checks its status (`status?`) and returns it (`instance!`). Notice that the DM is not bound to any particular cloud (Google or Azure), nor it is described in the LTS how one of them is actually chosen. We will show later on how a specific cloud selection policy can be enforced by the adaptation contract.

The behaviour of Google Compute Engine, is shown in Figure 1 (b). In Google's IaaS cloud, the operation for creating a VM is named `addInstance`, and the status of a machine can be checked with `getInstance`. Additionally, we assume that the cloud sends statistical information about CPU and memory

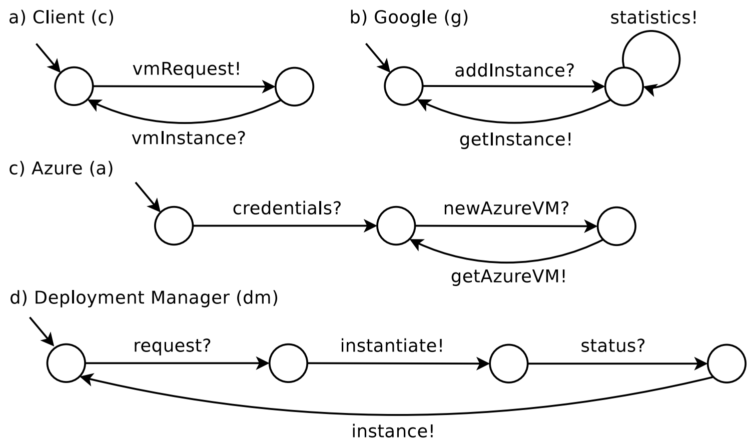

Fig. 1. LTSs describing the interfaces of the different peers

usage (`statistics!`). Figure 1 (c) shows the LTS corresponding to Windows Azure. We assume that some credentials must be received first (`credentials?`). Then, machine creation is performed with the operation `newAzureVM?`, while VM status is reported with `getAzureVM!`. The LTS describing the behaviour of a possible client, requesting several VMs, is shown in Figure 1 (a).

Finally, the vectors for composing the whole system, adapting the Client, the Deployment Manager, and Google's and Azure's clouds are as follows:

$$\begin{array}{lcl}
V_{request} & = & \langle \texttt{c:vmRequest!, dm:request?} \rangle \\
V_{instantiateG} & = & \langle \texttt{dm:instantiate!, g:addInstance?} \rangle \\
V_{statusG} & = & \langle \texttt{dm:status?, g:getInstance!} \rangle \\
V_{credentialsA} & = & \langle \texttt{a:credentials?} \rangle \\
V_{instantiateA} & = & \langle \texttt{dm:instantiate!, a:newAzureVM?} \rangle \\
V_{statusA} & = & \langle \texttt{dm:status?, a:getAzureVM!} \rangle \\
V_{instance} & = & \langle \texttt{c:vmInstance?, dm:instance!} \rangle
\end{array}$$

These vectors mostly show correspondence of events with different names. Note also that event `a:credentials?` has no correspondence in the DM, so it has no counterpart in the $V_{credentialsA}$ rule: this event will be issued by the adapter when required by the cloud. Additionally, a vector for event `g:statistics!` is omitted, since this event has no counterpart in the rest of the system. We can now automatically generate an adapter using Itaca's tools. The resulting adapter is shown in Figure 2.

3 Synchronizability of Adapted Systems

In this section, we present a few definitions characterizing the synchronizability property for adapted systems. The adapted synchronous composition of a set of

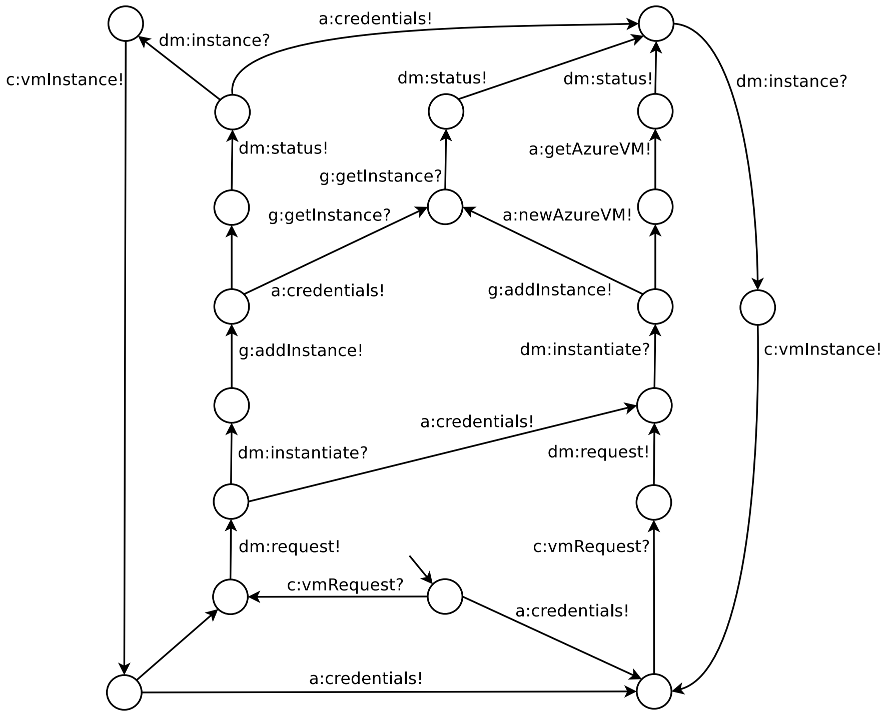

Fig. 2. Adapter LTS

peers corresponds to the system where a communication occurs when one peer can send (receive, *resp.*) an event and the adapter can receive (send, *resp.*) it.

Definition 4 (Adapted Synchronous Composition). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$ *and an adapter* $\mathcal{A} = (S, s^0, \Sigma, T)$, *their synchronous composition is the labelled transition system* $LTS_{as} = (S_{as}, s_{as}^0, \Sigma_{as}, T_{as})$ *where:*

- $S_{as} = S_1 \times \ldots \times S_n \times S$
- $s_{as}^0 \in S_{as}$ *such that* $s_s^0 = (s_1^0, \ldots, s_n^0, s^0)$
- $\Sigma_{as} = \cup_i \Sigma_i \cup \Sigma$
- $T_{as} \subseteq S_{as} \times \Sigma_{as} \times S_{as}$, *and for* $s = (s_1, \ldots, s_n, s_a) \in S_{as}$ *and* $s' = (s'_1, \ldots, s'_n, s'_a) \in S_{as}$ *we have that*

(p2a) $s \xrightarrow{m} s' \in T_{as}$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \Sigma^?$ *where* $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$, *and* $s_a \xrightarrow{m?} s'_a \in T$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k$

(a2p) $s \xrightarrow{m} s' \in T_{as}$ *if* $\exists j \in \{1, \ldots, n\} : m \in \Sigma^! \cap \Sigma_j^?$ *where* $\exists\ s_a \xrightarrow{m!} s'_a \in T$, *and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq j \Rightarrow s'_k = s_k$

(int) $s \xrightarrow{\tau} s' \in T_{as}$ *if* $\exists i \in \{1, \ldots, n\}$, $\exists\ s_i \xrightarrow{\tau} s'_i \in T_i$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k$ *and* $s'_a = s_a$

However, in an asynchronous scenario, peers communicate with the adapter asynchronously via FIFO buffers. Hence, each peer $\mathcal{P}_i$ is equipped with an unbounded input message buffer Q_i, and the adapter $\mathcal{A}$ with an input buffer Q. A peer can either send a message $m \in \Sigma^!$ to the tail of the adapter buffer Q at any state where this send message is available, read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head, or evolve independently through an internal τ transition. The adapter works in the same way. We recall that we focus on output events, since reading from the buffer is private non-observable information, which is encoded as an internal transition in the asynchronous system.

Definition 5 (Adapted Asynchronous Composition). *Given a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ with $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$, Q_i being its associated input buffer, and an adapter $\mathcal{A} = (S, s^0, \Sigma, T)$ with input buffer Q, their asynchronous composition is the labelled transition system $LTS_{aa} = (S_{aa}, s_{aa}^0, \Sigma_{aa}, T_{aa})$ where:*

- $S_{aa} \subseteq S_1 \times Q_1 \times \ldots \times S_n \times Q_n \times S \times Q$ *where* $\forall i \in \{1, \ldots, n\}$, $Q_i \subseteq (\Sigma_i^?)*$ *and* $Q \subseteq (\Sigma^?)*$
- $s_{aa}^0 \in S_{aa}$ *such that* $s_{aa}^0 = (s_1^0, \epsilon, \ldots, s_n^0, \epsilon, s^0, \epsilon)$ *(where ϵ denotes an empty buffer)*
- $\Sigma_{aa} = \cup_i \Sigma_i \cup \Sigma$
- $T_{aa} \subseteq S_{aa} \times \Sigma_{aa} \times S_{aa}$, *and for* $s = (s_1, Q_1, \ldots, s_n, Q_n, s_a, Q) \in S_{aa}$ *and* $s' = (s'_1, Q'_1, \ldots, s'_n, Q'_n, s'_a, Q') \in S_{aa}$ *we have that*

(p2a!) $s \xrightarrow{m!} s' \in T_{aa}$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \Sigma^?$, *(i)* $s_i \xrightarrow{m!} s'_i \in T_i$, *(ii)* $Q' = Qm$, *(iii)* $s'_a = s_a$, *(iv)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *and (v)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

(p2a?) $s \xrightarrow{\tau} s' \in T_{aa}$ *if* $m \in \Sigma^?$, *(i)* $s_a \xrightarrow{m?} s'_a \in T$, *(ii)* $mQ' = Q$, *(iii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : s'_k = s_k$

(a2p!) $s \xrightarrow{m!} s' \in T_{aa}$ *if* $\exists j \in \{1, \ldots, n\} : m \in \Sigma^! \cap \Sigma_j^?$, *(i)* $s_a \xrightarrow{m!} s'_a \in T$, *(ii)* $Q'_j = Q_j m$, *(iii)* $Q' = Q$, *(iv)* $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q'_k = Q_k$, *and (v)* $\forall k \in \{1, \ldots, n\} : s'_k = s_k$

(a2p?) $s \xrightarrow{\tau} s' \in T_{aa}$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$, *(i)* $s_i \xrightarrow{m?} s'_i \in T_i$, *(ii)* $mQ'_i = Q_i$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$, *(iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$, *(v)* $Q' = Q$, *and (vi)* $s'_a = s_a$

(int) $s \xrightarrow{\tau} s' \in T_{aa}$ *if* $\exists i \in \{1, \ldots, n\}$, *(i)* $s_i \xrightarrow{\tau} s'_i \in T_i$, *(ii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$ *(iv)* $Q' = Q$, *and (v)* $s'_a = s_a$

We use LTS_{aa}^k to define the *bounded adapted asynchronous composition*, where each message buffer is bounded to size k. The definition of LTS_{aa}^k can be obtained from Definition 5 by allowing send transitions only if the message buffer of the receiving peer has less than k messages in it. Otherwise, the sender is blocked, *i.e.*, we assume reliable communication without message losses.

The synchronizability property applies here by considering the adapter as a peer whose specificity is to interact with all the other participants.

Definition 6 (Branching Synchronizability). *A set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and an adapter* $\mathcal{A}$ *are branching synchronizable if* $\forall k \geq 1,\ LTS_{as} \equiv_{br} LTS^k_{aa}$.

It was proved that checking the equivalence between the synchronous composition and the 1-bounded asynchronous composition, *i.e.*, $LTS_{as} \equiv_{br} LTS^1_{aa}$, is a sufficient and necessary condition for branching synchronizability [26].

Theorem 1 (Deadlock-freeness). *A synchronizable system consisting of a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and an adapter* $\mathcal{A}$ *is deadlock-free when all participants communicate asynchronously via k-bounded or unbounded FIFO buffers.*

Proof. An adapter generated using synchronous synthesis techniques is deadlock-free (DF) according to [13, 24], that is, $LTS_{as} \models DF$. The system (set of peers interacting via an adapter) being synchronizable, we have $LTS_{as} \equiv_{br} LTS_{aa}$. Since branching equivalence preserves deadlock-freeness, if LTS_{as} is deadlock-free then LTS_{aa} is deadlock-free, that is, $LTS_{as} \models DF \Leftrightarrow LTS_{aa} \models DF$. $\square$

4 Asynchronous Adaptation

In this section, we present our approach for adapter synthesis when peers interact via FIFO buffers, its application to our running example, and a short overview of tool support.

4.1 Methodology

Figure 3 shows how our approach works for generating adapter LTSs in asynchronous environments. First of all, we apply compatibility checking techniques, *e.g.*, [26], for understanding whether the set of selected peers can be reused and composed directly, that is, without using adaptation techniques for compensating mismatch. If an adapter is required, the user needs to provide an adaptation contract. Note that this is the only step of our approach that requires human intervention. Given a set of peer LTSs and an adaptation contract, an adapter LTS is automatically synthesised by means of synchronous adapter generation techniques, *e.g.*, [13, 24]. 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 generation techniques relying on synchronous communication can be used as is in an asynchronous context.

If the system is not synchronizable, the user should refine the adaptation contract using the diagnostic returned by equivalence checking. 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. This latter case particularly arises when the adaptation contract enforces additional constraints by the use for instance of regex on vectors. It is worth emphasizing that these reasons for non-synchronizability can be used as guidelines when writing the adaptation contract. Keeping that in mind should help the user to converge more rapidly to a synchronizable system.

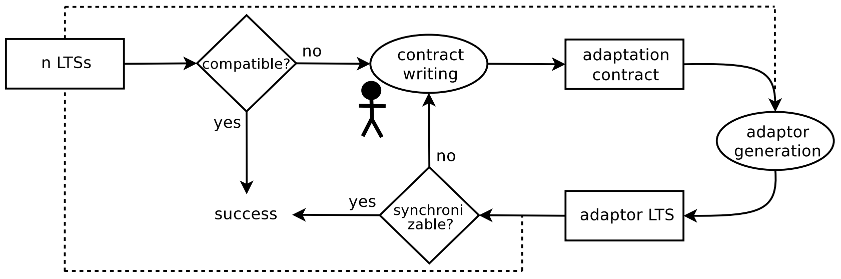

Fig. 3. Approach overview

4.2 Application to the Case Study

Let us go back to our multi-cloud running example. Given the participant LTSs and the set of vectors presented in Section 2, we can generate automatically the corresponding adapter. We first check synchronizability for this adapter composed with the peer LTSs. As a result, the verdict is false and we obtain the following counterexample: `c:vmRequest!`, `dm:request!`, `dm:instantiate!`, `g:addInstance!`, and `g:statistics!`, where the very last event appears in the asynchronous system but not in the synchronous one. Note that synchronizability checking focuses on emissions, hence the counterexample above contains only events sent by a peer to the adapter (`c:vmRequest!`, `dm:instantiate!`, `g:statistics!`) or by the adapter to a peer (`dm:request!`, `g:addInstance!`). This violation is due to the fact that the emission of `statistics` is not captured by a vector (yet), and this emission is inhibited in the synchronous system, while it is 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: $V_{statisticsG} = \langle$`g:statistics!`$\rangle$. The corresponding adapter (not shown here) is generated and it consists of 25 states and 41 states. We check again synchronizability and the system composed of the four peers interacting through the adapter is synchronizable, which means that the adapter can be used under asynchronous communication and the system will behave exactly the same whatever bound is chosen for buffers.

However, in this system the Deployment Manager may decide to instantiate always the same kind of VM (Google or Azure). If we want to enforce the DM to instantiate a VM in one of the two clouds alternatively, we have to impose a specific sequence in the application of vectors during the adapter generation process. This can be specified by means of the following regex:

$$\begin{array}{l}(\ V_{request} \ V_{instantiateG} \ V_{statisticsG} \ V_{statusG} \ V_{instance} \\ \quad V_{request} \ V_{credentialsA} \ V_{instantiateA} \ V_{statusA} \ V_{instance} \ ) \star\end{array}$$

The adapter generated using the aforementioned vectors and this regex contains 21 states and 24 transitions. However, when we check synchronizability with this new adapter, the check returns false with a counterexample containing 18 events. The problem in that case is that Windows Azure works in connected mode, in the sense that it receives credentials once, and then several new VM instances can be created. Nonetheless, the regular expression requires credentials to be submitted before any new instantiation. Therefore, when we arrive at the second instantiation of an Azure VM, the adapter submits again the `credentials!` event whereas the Azure peer does not expect it: this emission appears in the asynchronous system but not in the synchronous version.

To solve this problem, we need to relax the regular expression by avoiding the strict sequence of vectors. A simple idea is to say that after each request, we can execute in any order the vectors where one of the two VM providers (Google or Azure) is involved. This discards the problem encountered with the credentials, whose rule is executed the first time only. Here is the new regex:

$$\begin{array}{l}(\ V_{request} \ (\ V_{instantiateG} \mid V_{statisticsG} \mid V_{statusG} \mid V_{instance} \ ) \star \\ \quad V_{request} \ (\ V_{credentialsA} \mid V_{instantiateA} \mid V_{statusA} \mid V_{instance} \ ) \star \ ) \star\end{array}$$

The corresponding adapter consists of 32 states and 38 transitions. When we check the synchronizability with this new adapter, the system is synchronizable and accordingly this adapter can be used under asynchronous communication.

4.3 Tool Support

Tool support consists of two parts: a set of tools for generating adapters and some automated techniques for checking synchronizability. As for adapter synthesis, we reuse the Itaca toolbox [9]. Itaca takes as input a set of Symbolic Transition Systems (LTSs are STSs without message parameters) and an adaptation contract, and generates an adapter LTS, from which BPEL code can be generated, see [24] for details. As for synchronizability, we rely on process algebra encodings and equivalence checking. More precisely, we developed a Python script, which generates for all input LTSs (peers and adapter) some code in the LNT process algebra [15]. Then, we use the CADP toolbox [17], which accepts LNT specifications as input. Particularly, we rely on CADP exploration tools for computing the required (synchronous/asynchronous) compositions and CADP equivalence checker for checking synchronizability.

Table 1 presents experimental results for some real-world examples. The table gives for each example the number of peers (P), the total number of states (S) and transitions (T) involved in these examples, the size of the 1-bounded asynchronous system (minimised modulo branching reduction), and the overall time for checking synchronizability (including composition generations, minimisations, and equivalence checking). It is worth noting that out of these 14 examples, 7 of the adapters generated for synchronous communication can be directly reused as they are in asynchronous environments, while 7 require to use our approach in order to replay adapter synthesis techniques until obtaining an adapter which satisfies synchronizability. Computation times are quite short since all the examples found in the literature are quite small.

Example	|P|+1	$\|S\|/\|T\|$	LTS_a^1 ($\|S\|/\|T\|$)	Synchro.	Time
FTP Transfer [6]	3	20/17	13/15	×	52s
Client/Server [12]	3	14/13	8/7	√	54s
Mars Explorer [8]	3	34/34	19/22	×	49s
Online Computer Sale [16]	3	26/26	11/12	√	53s
E-museum [13]	3	33/40	47/111	×	53s
Client/Supplier [10]	3	31/33	17/19	√	49s
Restaurant Service [1]	3	15/16	10/12	√	55s
Travel Agency [30]	3	32/38	18/21	√	52s
Vending Machine [18]	3	15/14	8/8	√	49s
Travel Agency [4]	3	42/57	23/34	×	45s
Client/Server [31]	4	19/24	18/32	×	64s
SQL Server [29]	4	32/38	20/27	×	62s
SSH Protocol [22]	4	26/28	16/18	√	56s
Booking System [23]	5	45/53	27/35	×	85s

Table 1. Experimental results

5 Related Work

In this section, we present the most relevant and recent results in the software adaptation area. First of all, notice that adaptation differs from automatic software composition approaches, particularly studied in the Web services area (see, *e.g.*, [21, 5]), where services involved into a new composition are assumed to perfectly match altogether with respect to certain compatibility property.

Van der Aalst *et al.* [1] propose a solution to behavioural adaptation based on open nets, a variant of Petri nets. Their generation algorithm produces an adapter which is obtained through several steps. First, a message transformation net, called engine, is generated from a set of message transformation rules. Then, a behavioural controller (a transition system) is synthesised for the product net of the services and the engine. Adapters are finally implemented in the BPEL orchestration language. In [25], the authors identify several kinds of mismatch between Web service interfaces. They provide a method for identification of the split/merge class of interface mismatch and a semi-automated, behaviour-aware approach for interface-level mismatch that results in identifying parameters of mapping functions that resolve that mismatch. They use and extend approaches in ontology matching for static matching of service interfaces to identify split/merge mismatch. In addition, they propose depth-based and iterative reference-based approaches that incorporate behavioural information during interface matching.

In [13, 24], the authors proposed automated techniques for generating an adapter model from a set of service interfaces and a contract. The approach relies on an encoding into process algebra together with on-the-fly exploration and reduction techniques. Verification of contracts is also possible by using model checking techniques. Last, code is automatically generated from the adapter model to BPEL, which may finally be deployed. Inverardi and Tivoli [19] formalise a method for the automated synthesis of modular connectors. A modular connector is structured as a composition of independent mediators, each of them corresponding to the solution of a recurring behavioural mismatch. The paper proves that the connector decomposition is correct and shows how it promotes connector evolution on a case study. Bennaceur *et al.* [4] propose a technique for automated synthesis of mediators using a quotient operator, that is based on behavioural models of the components and an ontological model of the data domain. The method supports both off-line and run-time synthesis. The obtained mediator is the most general component that ensures deadlock-freedom and the absence of communication mismatch.

It is worth observing that, although all these papers present interesting approaches tackling software adaptation from different points of view, they assume that peers interact synchronously. There were a few attempts to generate adapters considering asynchronous communication. Padovani [27] presents a theory based on behavioural contracts to generate orchestrators between two services related by a subtyping (namely, sub-contract) 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 on the orchestrator, namely regularity and contractivity. However, this work does not support name mismatch nor data-related adaptation. Seguel *et al.* [30] 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 [18] 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.

6 Conclusion

Software adaptation is the only solution for building new systems by combining black-box services that are relevant from a functional point of view, but do not exactly match one with another and therefore require adjustments during the composition process. Most existing approaches 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 highly complicates the synthesis process because we may have to face infinite systems when generating the adapter behaviour. Our approach provides a solution to this problem by using the synchronizability property and adapter generation techniques for synchronous communication. This enables us to propose an iterative approach for synthesising adapters, which work properly in asynchronous environments. Our approach is tool-supported and has been applied to a large variety of real-world examples found in the literature.

Acknowledgements

This work has been partially funded by the Spanish Government under project TIN2012-35669, and by the University of Málaga, Campus de Excelencia Internacional, Andalucía Tech.

References

1. van der Aalst, W.M.P., Mooij, A.J., Stahl, C., Wolf, K.: Service Interaction: Patterns, Formalization, and Analysis. In: Proc. of SFM'09. LNCS, vol. 5569, pp. 42–88. Springer (2009)
2. de Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proc. of ESEC/FSE'01. pp. 109–120. ACM Press (2001)
3. Basu, S., Bultan, T., Ouederni, M.: Deciding Choreography Realizability. In: Proc. of POPL'12. pp. 191–202. ACM (2012)
4. Bennaceur, A., Chilton, C., Isberner, M., Jonsson, B.: Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning. In: Proc. of SEFM'13. LNCS, vol. 8137, pp. 274–288. Springer (2013)
5. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of Web services via planning in asynchronous domains. Artificial Intelligence 174(3-4), 316–361 (2010)
6. Bracciali, A., Brogi, A., Canal, C.: A Formal Approach to Component Adaptation. Journal of Systems and Software 74(1), 45–54 (2005)
7. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. Journal of the ACM 30(2), 323–342 (1983)
8. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Proc. of ICSOC'06. LNCS, vol. 4294, pp. 27–39. Springer-Verlag (2006)
9. Cámara, J., Martín, J.A., Salaün, G., Cubo, J., Ouederni, M., Canal, C., Pimentel, E.: ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In: Proc. of ICSE'09. pp. 627–630. IEEE (2009)
10. Cámara, J., Martín, J.A., Salaün, G., Canal, C., Pimentel, E.: Semi-Automatic Specification of Behavioural Service Adaptation Contracts. Electr. Notes Theor. Comput. Sci. 264(1), 19–34 (2010)

11. Canal, C., Murillo, J.M., Poizat, P.: Software Adaptation. L'Objet 12(1), 9–31 (2006)
12. Canal, C., Poizat, P., Salaün, G.: Synchronizing Behavioural Mismatch in Software Composition. In: Proc. of FMOODS'06. LNCS, vol. 4037, pp. 63–77. Springer-Verlag (2006)
13. Canal, C., Poizat, P., Salaün, G.: Model-Based Adaptation of Behavioural Mismatching Components. IEEE Trans. on Software Engineering 34(4), 546–563 (2008)
14. Canal, C., Salaün, G.: Adaptation of Asynchronously Communicating Software. In: Proc. of ICSOC'14. LNCS, vol. 8831, pp. 437–444. Springer (2014)
15. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4) (2011), INRIA/VASY, 149 pages
16. Cubo, J., Salaün, G., Canal, C., Pimentel, E., Poizat, P.: A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In: Proc. of FACS'07. ENTCS, vol. 215, pp. 39–55. Elsevier (2007)
17. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Proc. of TACAS'11. LNCS, vol. 6605, pp. 372–387. Springer (2011)
18. Gierds, C., Mooij, A.J., Wolf, K.: Reducing Adapter Synthesis to Controller Synthesis. IEEE T. Services Computing 5(1), 72–85 (2012)
19. Inverardi, P., Tivoli, M.: Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In: Proc. of ICSE'13. pp. 3–12. IEEE / ACM (2013)
20. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures, pp. 35–49. Kluwer Academic Publishers (1999)
21. Marconi, A., Pistore, M.: Synthesis and Composition of Web Services. In: Proc. of SFM'09. LNCS, vol. 5569, pp. 89–157. Springer (2009)
22. Martín, J.A., Pimentel, E.: Contracts for Security Adaptation. J. Log. Algebr. Program. 80(3-5), 154–179 (2011)
23. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In: Proc. of ICSOC'08. LNCS, vol. 5364, pp. 84–99. Springer (2008)
24. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. IEEE Trans. on Software Engineering 38(4), 755–777 (2012)
25. Nezhad, H.R.M., Xu, G.Y., Benatallah, B.: Protocol-Aware Matching of Web Service Interfaces for Adapter Development. In: Proc. of WWW'10. pp. 731–740. ACM (2010)
26. Ouederni, M., Salaün, G., Bultan, T.: Compatibility Checking for Asynchronously Communicating Software. In: Proc. of FACS'13. LNCS, Springer (2013)
27. Padovani, L.: Contract-Based Discovery and Adaptation of Web Services. In: Proc. of SFM'09. LNCS, vol. 5569, pp. 213–260. Springer (2009)
28. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Trans. on Software Engineering 28(11), 1056–1076 (2002)
29. Poizat, P., Salaün, G.: Adaptation of Open Component-based Systems. In: Proc. of FMOODS'07. LNCS, vol. 4468, pp. 141–156. Springer (2007)
30. Seguel, R., Eshuis, R., Grefen, P.W.P.J.: Generating Minimal Protocol Adaptors for Loosely Coupled Services. In: Proc. of ICWS'10. pp. 417–424. IEEE Computer Society (2010)

31. Tivoli, M., Inverardi, P.: Failure-Free Coordinators Synthesis for Component-Based Architectures. Sci. Comput. Program. 71(3), 181–212 (2008)
32. Yellin, D.M., Strom, R.E.: Protocol Specifications and Components Adaptors. ACM Trans. on Programming Languages and Systems 19(2), 292–333 (1997)