Stability-based Adaptation of Asynchronously Communicating Software

Carlos Canal[1] and Gwen Salaün[2]

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

Abstract. Software Adaptation aims at composing incompatible black-box components or services (peers) whose individual functionality is as required for the new system. Adaptation techniques aim at automatically generating new components called adapters. An adapter works as an orchestrator and makes the involved peers work correctly together by receiving all messages exchanged in the system and by correcting mismatch between them. A challenging issue in this area is to consider that peers are described with (possibly cyclic) behavioural models and interact asynchronously, that is, exchanging messages via message buffers. The synthesis of adapters in this context is difficult because the composition of peers may result in infinite systems. In this paper, we propose new adaptation techniques, which rely on a property of communicating systems called stability. Stability aims at verifying whether a communicating system exhibits the same observational behaviour from a certain buffer bound on. We also provide adapter generation techniques using process algebra encodings and enumerative analysis techniques.

1 Introduction

New software is constructed in many cases by reusing and composing existing software elements,hereafter called *peers*. These peers correspond to a large variety of software, such as software components, Web servers, databases, Graphical User Interfaces, Software-as-a-Service in the cloud, or Web services. The composition of such heterogeneous software pieces is possible because peers are equipped with public interfaces, which exhibit their provided/required services as well as any other composition requirements that must be respected to ensure the correct execution of the system. A problem in this context is that some peer may be relevant *wrt.* a new composition-to-be from a functional point of view, but does not exactly match with the partner peers from an interface point of view. Mismatch takes different forms such as disparate operation names or unspecified message receptions, and it prevents the direct assembly of the peers.

Software Adaptation [26, 7] is a non-intrusive solution for composing black-box software peers that present interface mismatch, leading to deadlock or other undesirable behaviour when they 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. The adapter acts as an orchestrator and makes the involved peers work correctly together by receiving all messages and by compensating for mismatch. Many solutions have been proposed since the seminal work by Yellin and Strom [26], but most of them assume that peers interact using synchronous communication, that is, synchronization via rendez-vous.

One of the main open challenges in the adaptation area is to assume that peers interact using asynchronous communication, which is a valid assumption given that nowadays many systems rely on this communication model (cloud computing, Web, grid computing, GALS, multi-core architectures, IoT, etc.). 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 is undecidable for communicating finite state machines [4]. An option is to arbitrary bound the sources of infiniteness (buffers, cycles, number of participants, etc.), but we want to avoid imposing this kind of constraints, since it would unnecessarily restrict the behaviour of the whole system.

We assume that peers are modelled using behavioural descriptions and interact asynchronously via (possibly unbounded) FIFO buffers. In a previous work [9], we presented a preliminary proposal for asynchronous adaptation in which a sufficient condition, called *synchronizability*, was required. However, many asynchronous systems are not synchronizable. Hence, in order to widen the number of systems to be adapted, in this paper we propose new synthesis techniques, which rely on an encoding into the LNT process algebra [10] on the one hand, and on a property of *stability* [1] on the other hand. Using stability is an improvement over synchronizability, as many systems in practice are not synchronizable yet stable. A set of peers is stable if from some buffer bound k, the k-bounded asynchronous composition is equivalent to the $k+1$-bounded asynchronous composition, considering only the ordering of output messages and ignoring that of input messages. If this k exists, it is proved [1] that the observable behaviour remains the same for any larger buffer bound. This property can be verified using equivalence checking techniques on finite systems, although the set of peers interacting asynchronously can result in infinite systems. Based on this result, one can check on the system a property, concerning output messages, for the smallest bound satisfying stability and claim that this property is also satisfied for any larger bound. We use stability here for verifying whether an adapter generated for a certain bound k can be used with any larger bound, or even with unbounded buffers.

As far as the adapter synthesis techniques are concerned, we encode all inputs (peers, contract, buffers) into the LNT process algebra and use the CADP verification toolbox [13] for generating the corresponding adapter model. The stability property is also checked automatically using the CADP equivalence checker. Since the adaptation contract is manually written, the designer can take advantage of the LNT encoding to verify using CADP that the final adapter works correctly, that is, respects certain properties of interest. We have validated our

approach on several case studies, one of them presented in detail throughout this paper.

The rest of this paper is organized as follows. Section 2 introduces the behavioural model for peers and the notation for specifying adaptation contracts. Section 3 overviews the encoding into LNT. In Section 4, we present the stability property and how we use it in our context. Section 5 shows how we generate the adapter model from the LNT encoding and using CADP verification techniques. In this section, we also present our whole adaptation method for asynchronous environments. Finally, Section 6 surveys related work, and Section 7 concludes this paper.

2 Models

In this section, we first introduce the interface model through which peers are accessed and used. Then, we define adaptation contracts, and present the motivating example that will be used throughout the paper.

2.1 Interface LTS

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,* $\Sigma = \Sigma^! \cup \Sigma^?$ *is a finite alphabet partitioned into a set* $\Sigma^!$ *(*$\Sigma^?$*, resp.) of send (receive, resp.) message events, 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 a message event $p? \in \Sigma^?$ in the alphabet of the LTS describing the behaviour of the peer, and for each operation r required from its environment, there is a message event $r! \in \Sigma^!$. When two peers present events with the same name and complementary directions ($a!$, $a?$) they can be matched for inter-peer communication through message-passing.

Note that as usually done in the literature [16, 11, 23], 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) [18] instead of LTSs. However, this renders the definitions and results presented in this work much longer and cumbersome, without adding anything substantial to the technical aspects of our proposal. Hence, it is omitted in this paper.

It is worth observing that other formalisms, such as process algebra, could be used alternatively to LTS [8]. However, for this paper we have preferred to use LTS as the input notation of our proposal, since they provide a compact representation, graphical, and easy to understand for all developers.

2.2 Adaptation Contracts and Adapter LTS

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 an adaptation contract [8]. Adaptation contracts consist of rules that 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, we prefix event names with 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.

In some complex adaptation scenarios, adaptation rules must be taken contextually (*i.e.*, vectors cannot be applied at any time, but only in certain situations). For this purpose, we may use regular expressions (regex) on vectors [9], indicating a pattern for applying them that will constrain the adaptation process, enforcing additional properties on the adapter. This endows adaptation contracts with extended expressivity, though in this work we do not show their use, in order to avoid additional complexity in the presentation of our proposal.

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.*

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. However, the adaptation methodology that we propose is iterative, which helps in writing the contract contract. Furthermore, in [5, 6], we presented a tool-supported approach for assisting and making easier the specification of the adaptation contract. For more details on adaptation contracts and the kinds of mismatch that can be resolved with them, we refer to [8].

Given a set of peers represented by their LTS interfaces and an adaptation contract, our goal is to generate an adapter, which will play the role of *man-in-the-middle*, solving the mismatch presented by the peers. The adapter is also represented by an LTS consisting of messages to be consumed from its buffer and messages to be sent to the other peers. The adapter also keeps track of the messages received by its own local buffer. This information is important to enforce the adapter to execute the correct behaviour, avoiding engaging in a branch that may lead to an erroneous execution of the whole system.

Definition 4 (Adapter LTS). *An adapter LTS is a tuple* (S, s^0, Σ, T) *where:* S *is a set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^? \cup \Sigma^B$ *is a finite alphabet partitioned into a set* $\Sigma^!$ *(*$\Sigma^?$*, resp.) of send (receive, resp.) messages and a set of messages received by its buffer* Σ^B*, and* $T \subseteq S \times \Sigma \times S$ *is the transition function.*

In the following we will show how this adapter LTS can be automatically generated from the LTS interfacs of the peers and the adaptation contract.

2.3 Running example

In order to illustrate the main features of our proposal, the following motivating example will be used throughout this paper. Consider a simple Client/Server system, in which clients are identified to the server by their username and password, and submit requests for a certain service. Upon receiving the result of the request, the client issues an acknowledging message, and then quits. The interface LTS representing the behaviour of clients is shown in Figure 1, top, where the black dot indicates the initial state.

On the other side, the server follows a similar behaviour, as shown in Figure 1, left bottom. The main differences, which are used in the example in order to show how to perform adaptation, are: *(i)* connections are expected by the server with a single *login?* message (instead of two separate *usr!* and *pwd!* messages issued by the client); *(ii)* after login, the server is ready to receive several consecutive requests or a logout message, while the client only sends one request before quitting; and *(iii)* messages for disconnection are also named differently in the client and the server. Finally the server interacts with a third peer, a database log (Figure 1, right bottom), which stores all the requests fulfilled by the server.

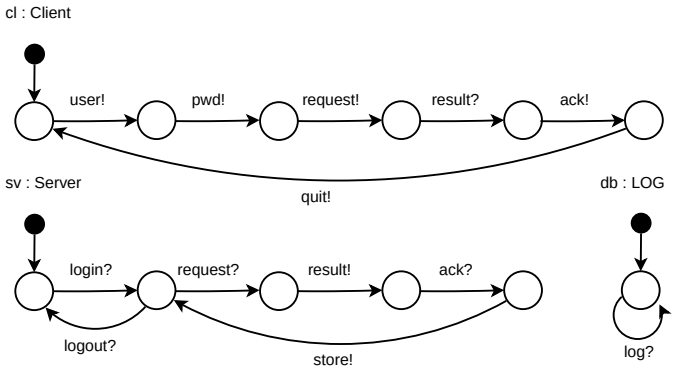

Fig. 1. Interface LTSs of the peers.

The example above has been chosen deliberately simple in order to avoid it taking too much space in this paper. However, it shows the different kinds of adaptation that our proposal addresses: differences in message names (*e.g.*, *quit!* and *logout?*), differences in the granularity or the order of messages (*e.g.*, *user!*, *pwd!* and *login?*), and differences in decision-taking roles and other higher level behavioural aspects (*e.g.*, the client decides to quit after a request, while the server allows new requests). How to address all these differences or mismatch between the interfaces of the peers is specified with an adaptation contract. Assuming that *sv* refers to a server, *cl* to its client and *db* to the database log, these three peers can be adapted by means of the following contract, which shows how message names are interconnected between the different peers involved in this system:

$$
\begin{array}{l}
\{\ \langle cl{:}user!,\ sv{:}login?\rangle,\\
\ \ \langle cl{:}pwd!\rangle,\\
\ \ \langle cl{:}quit!,\ sv{:}logout?\rangle,\\
\ \ \langle cl{:}request!,\ sv{:}request?\rangle,\\
\ \ \langle cl{:}result?,\ sv{:}result!\rangle,\\
\ \ \langle cl{:}ack!,\ sv{:}ack?\rangle,\\
\ \ \langle sv{:}store!,\ db{:}log?\rangle\ \}
\end{array}
$$

3 Process Algebra Encoding

Our asynchronous adapter generation techniques rely on an encoding into LNT [10] that we overview in this section. LNT is a formal specification language which combines traits of process calculi, functional and imperative languages. We chose LNT for two main reasons. First, it is expressive enough for encoding all inputs (LTSs, contract, buffers, architecture) of our problem. Second, it is equipped with a rich verification toolbox (CADP) that we use for checking the existence of an adapter and, if this is the case, for generating an adapter LTS and for analyzing properties of interest on it.

Interface LTSs. An LNT process is generated for each state in the interface LTS of a peer. The alphabet of the process contains the set of messages appearing in the transitions of the LTS. The behaviour of the process encodes all the transitions of the LTS going out from the corresponding state. If there is no such transition, the body of the process is the null statement. If there is a single transition, the body of the process corresponds to the message labelling this transition, followed by a call to the process encoding the target state of the transition. If there is more than one transition, we use the select operator, which encodes a nondeterministic choice between the different transitions going out of that state. Name clashes are avoided by prefixing each message with the name of the corresponding peer. We encode emitted messages (received messages, resp.) with a _EM (_REC, resp.) suffix. These suffixes are necessary because LNT symbols ! and ? are used for data transfer only.

Adaptation Contract. The vectors are encoded into an LNT process called *contract*. The process alphabet is composed of all received and emitted messages between the adapter-to-be and the involved participant peers, that is, all messages appearing in the vectors. Each vector is encoded as a sequence of actions starting with the emissions and followed by the receptions. Notice that in the LNT process representing the adaptation contract, message directions are reversed with respect to the peers because the adapter will receive the output messages and emit the input messages expected by the recipient peers.

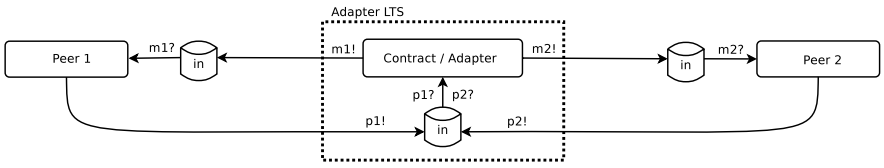

Fig. 2. Architecture and exchanged messages.

Asynchronous assembling. Now, we need to encode how all participants (peer interfaces and adaptation contract) are composed together using asynchronous communication. The architecture of the whole assembly is shown in Figure 2. The contract represents an abstract description of the future adapter, and all messages must go through this adapter, which acts as a centralized orchestrator. Each participant is equipped with an input FIFO buffer. A buffer in LNT is first encoded using an LNT list and LNT functions are used to describe classic operations on these buffers (*e.g.,* adding and retrieving messages). Then, for the behavioural part, a buffer is encoded using a process with a buffer data type as parameter. This process can receive messages from the other participants, and it synchronizes with its own participant when the latter wants to read a message. More precisely, when a participant reads a message, it reads the oldest message in its buffer. When a participant sends a message to another participant, it sends the message to the input buffer of that participant. In the next sections, we will show how buffer bounds are determined for generating the adapter LTS while avoiding the manipulation of infinite state spaces.

We also generate a process encoding each couple (*participant*, *buffer*) that corresponds to a parallel composition (par) of the participant with its buffer. The synchronization set contains messages consumed by the participant from its buffer. Finally, the whole system (main process in LNT) consists of the parallel composition of all these couples. It is worth noting that since the involved peers communicate via the adapter, they evolve independently from one another and are therefore composed using the par operator without synchronizations. In contrast, the couple (*contract*, *buffer*) must synchronize with all couples

(*peer*, *buffer*) on all emissions from/to the peers, and this is made explicit in the corresponding synchronization set of this parallel composition.

4 Stability of Adapted Systems

In this section, we characterize the stability property for adapted systems, where peers communicate with the adapter asynchronously via FIFO buffers. Hence, each peer $\mathcal{P}_i$ is equipped with an 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, or either read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head. 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)* $s'_a = s_a$

We denote by LTS_{aa} an unbounded adapted asynchronous composition, while we use LTS_{aa}^k for referring to the *k-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 stability property applies here by considering the adapter as a peer whose peculiarity is to interact with all the other participants.

Definition 6 (Stability). *A set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and an adapter* $\mathcal{A}$ *are stable if* $\exists k$ *such that* $LTS^k_{aa} \equiv_{br} LTS^q_{aa}$ $(\forall q > k)$.

A sufficient condition for ensuring stability was presented in [1]: if there exists a bound k such that the k-bounded and the $(k+1)$-bounded asynchronous systems are branching equivalent, *i.e.*, $LTS^k_{aa} \equiv_{br} LTS^{k+1}_{aa}$, then the system remains stable, meaning that the observable behaviour is always the same for any bound greater than k. The smallest k satisfying the stability property can be found using heuristics and a search algorithm. However, stability is undecidable. Therefore an arbitrary max bound is used during these computations and the algorithm stops when the current value goes beyong that arbitrary value. In that case, stability checking is inconclusive.

5 Adapter Generation and Methodology

In the previous section we have defined stability for adapted asynchronous systems. If a system is stable for a certain bound k, we are able to generate an adapter model that communicates asynchronously with the peers, where all the participants use buffers of size k. The adapter will play the role that until now had taken the adaptation contract. This adapter is obtained from our LNT encoding by keeping only the behaviour we expect from the adapter point of view, that is, we need to preserve send and receive messages for the adaptation contract. To do so, we hide message exchanges corresponding to consumptions of the peers from their buffers and we rename emissions from peers to the adaptation contract (Σ^B) in order to distinguish these messages from the adapter regular behaviour ($\Sigma^!$ and $\Sigma^?$). In order to keep only the behaviour corresponding to the most permissive adapter, we use CADP compilers, which explore all the possible executions of the generated LNT specification. We also make use of minimization techniques available in CADP for eliminating all internal actions, removing duplicated paths, and determinizing the final LTS. The whole adapter generation process is achieved automatically.

Figure 2 shows an example of architecture with the contract/adapter and two peers. Each participant is equipped with one input buffer. The dashed box shows the messages we keep in order to generate the adapter LTS where $m_i! \in \Sigma^!$, $p_i? \in \Sigma^?$, and $p_i! \in \Sigma^B$, $i \in \{1, 2\}$.

Note that stability is checked on the whole LNT encoding. We show below that this property is preserved when we extract the adapter LTS from this encoding for using it with the peer LTSs, all interacting via FIFO buffers.

Theorem 1 (Stability preservation). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *and an adaptation contract* V*, if the corresponding asynchronous LNT encoding is stable for a certain* k*, then the system where all peers interact through the generated adapter LTS A via* k*-bounded FIFO buffers is also stable for this* k*.*

Proof. Let $S_1^k = ((\mathcal{P}_1|Q_1)|\ldots|(\mathcal{P}_n|Q_n))|(V|Q)$ be the encoding into LNT of the peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, of the contract V, and of FIFO buffers Q_i for peers and Q for the contract/adapter-to-be. The alphabet $\Sigma_V = \Sigma_V^! \cup \Sigma_v^?$ of V coincides with the alphabet $\Sigma = \Sigma_A^! \cup \Sigma_A^? \cup \Sigma_A^B$ of the adapter LTS A, that is, $\Sigma_V^! = \Sigma_A^!$ and $\Sigma_V^? = \Sigma_A^?$, but for actions Σ_A^B. However, actions in Σ_A^B are not synchronized with the system, they are internally used in the adapter LTS for keeping track of the content of its buffer Q only.

Once the adapter is generated, the current behaviour of the whole system is as follows: $S_2^k = ((\mathcal{P}_1|Q_1)|\ldots|(\mathcal{P}_n|Q_n))|(A|Q)$. Actually, the adapter LTS A is obtained by extraction from S_1^k, by keeping the behaviour of V constrained by the peers' behaviours as explained beforehand in this section, which is exactly the behaviour of A. Thus, $(C|V) \equiv_{br} (C|A)$, where C stands for the context, *i.e*, the rest of the system. Hence, $S_1^k \equiv_{br} S_2^k$ and this proves the theorem. ■

Figure 3 gives an overview of our approach for generating an adapter LTS in asynchronous environments. First of all, we assume that the peers are incompatible and thus cannot be reused and composed directly without using adaptation techniques for compensating mismatch. This can be checked using existing compatibility techniques as those presented, *e.g.*, in [21]. If an adapter is required, the user needs to provide an adaptation contract. The next step consists in encoding all these inputs (peer LTSs and adaptation contract) into LNT as presented in Section 3.

Then, we check stability directly on the LNT encoding, trying to find a k from which the k-bounded adapted asynchronous composition and the $k+1$-bounded adapted asynchronous composition are equivalent. If this is the case, it means that the system is stable and its observable behaviour will remain the same whatever bound is chosen for buffers from that bound k. In that case, we can generate the adapter for that k, and it can be used in practice for whatever bound equal or greater than it. If the system is not stable, the sole solution is to fix an arbitrary bound before generating the adapter model, to generate the adapter LTS for that bound, and to use it further with that bound only.

Finally, since the adaptation contract is written manually, some mistake may appear at this level ending up with a faulty adapter. However, we can take advantage of the LNT encoding in order to formally analyse the system. This can be achieved by verifying the global LTS obtained direclty from the encoding and corresponding to the execution of the whole application (peers and adapter), or by verifying the adapter LTS obtained after synthesis from the LNT encoding as explained at the beginning of this section. In both cases, one can use the verification techniques and tools available in the CADP toolbox, and in particular, the Evaluator model checker, which accepts as input an LTS and an MCL formula [19], and returns a diagnostic (Boolean value + a counterexample if the property is false). If some property is not satisfied, we can go back to the contract writing, make corrections on it, and start again from this step the overall synthesis.

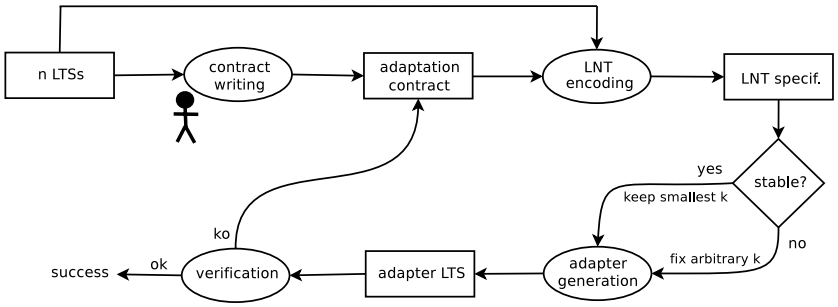

Fig. 3. Overview of our approach.

Coming back to our Client/Server example, we start from the LTSs of the peers, as presented in Section 2.3. As already explained there, the client, the server, and the database log show several sources of mismatch, the most obvious being that they communicate using different message names, but also that messages do not correspond one-to-one between the server and the client (during the login phase), and that the server admits several requests after connection, while the client does not. Hence, adaptation is required, and the adaptation contract presented in Section 2.3 is the first step of the adaptation process.

Then, we can check whether the system is synchronizable. If that were the case, we would be under the conditions defined in [9], which require the asynchronous system to be behaviourally equivalent to its synchronous version. This is not the case of our adapted Client/Server system, in which the client is able to issue several output messages (*user!*, *pwd!*, *request!*) in a row in the asynchronous version of the system, whereas this is not possible in the synchronous system because the adapter, after the reception of the two first messages (*user?*, *pwd?*), cannot receive the third one (*request?*) until it sends the *login!* message to the server. Hence, the results in [9] do not apply to our example, whereas the approach presented in this paper works as we will show in the rest of this section.

First of all, we need to check whether the system is stable. In order to analyse stability, both the interface LTSs of the peers and the adaptation contract are automatically encoded into LNT, as described in Section 3, and we check the LNT resulting system as defined in Section 4. We use the CADP toolbox for checking this property, which finds out that the asynchronous adapted system is stable for $k = 4$. Intuitively, this means than from buffers of that size, the observable collective behaviour of all peers remain the same, and hence, we can generate an asynchronous adapter using buffers bounded to that size. For this particular example, the asynchronous adapter presents 1,630 states and 4,278 transitions, though its generation takes only a few seconds. These figures show that, despite we have committed to a very simple system, asynchronous adaptation could not be possibly performed without automated techniques, as those presented in this paper. The asynchronous adapter, generated and visualized

with CADP, is shown in Figure 4, after the removal of internal transitions and identical paths, and abstracting for messages in Σ_B in order to make it fit in one page. We remind that event names in the adapter are reversed with respect to those of the peers and the adaptation contract, as explained in Section 3.

Once we have obtained the adapter, we can check the system (adapter alone or composition of the adapter with the peers) for properties of interest, like for instance deadlock freedom. Not fulfilling these properties would mean that the adaptation contract is ill-written, and from the counterexample provided we could adjust the contract, until the system behaves as expected. In our case, the adapted system is deadlock free, and we can also check for additional user-defined properties, which are expected to be enforced by the adapter. For instance, that every *request!* of the client will be followed by the delivery of the *result!* message, or that every *login?* message received by the server is followed by a corresponding *logout?* message. All these properties can be automatically analysed using the CADP model checker. When all properties of interest are satisfied, and this is the case with our example, we can conclude that our adaptation problem is solved.

6 Related Work

First of all, adaptation differs from automatic software composition approaches, particularly studied in the Web services area, *e.g.*, [17, 3], where services involved into a new composition are assumed to perfectly match altogether with respect to certain compatibility property [12].

The major part of the contributions on the software adaptation area assume that peers interact synchronously, while our proposal addresses asynchronous communication. Van der Aalst *et al.* [25] propose a solution to behavioural adaptation based on open nets, a variant of Petri nets. A behavioural controller (a transition system and BPEL) is synthesised for the product net of the services and a set of message transformation rules. In [20], the authors 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. In [8, 18], the authors proposed automated techniques for generating an adapter model from a set of service behavioural interfaces and an adaptation contract. Some BPEL code is automatically generated from the adapter model, which may finally be deployed.

Inverardi and Tivoli [15] 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. Bennaceur *et al.* [2] 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 obtained mediator is the most general component that ensures deadlock-freedom and the absence of communication mismatch.

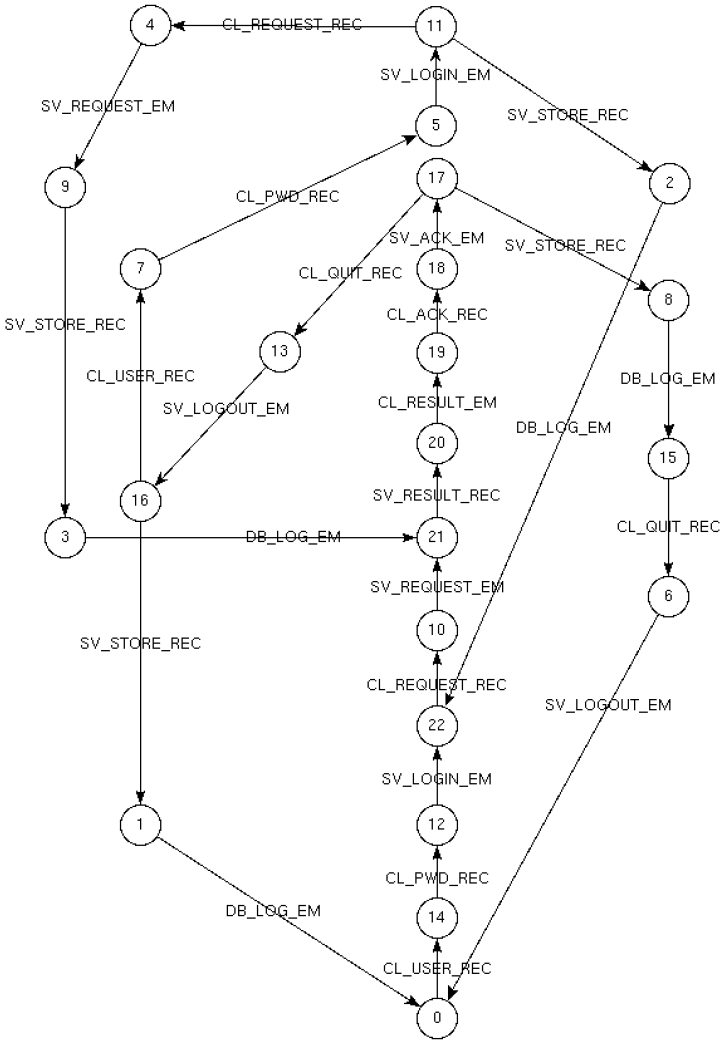

Fig. 4. Asynchronous adapter for the running example.

There are only a few attempts to generate adapters considering asynchronous communication. Padovani [22] 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.* [24] 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 [14] 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.

In [9], we presented a solution to the software adaptation problem by using the synchronizability property and adapter generation techniques for synchronous communication. The adapter synthesis in this approach relies on an iterative process, which works properly in asynchronous environments. The main limitation of our previous work is that the synchronizability property is quite restrictive and requires asynchronous systems to behave *de facto* as synchronous. Stability is a much looser condition, allowing to address a wider class of asynchronous systems.

7 Conclusion

Software adaptation is an approach for simplifying the reuse of existing peers when building a new software by composition of these entities. Adaptation is particularly relevant when the peers to be composed fulfill the functional requirements of the system but they are not compatible from an interface point of view. In that case, we can rely on such techniques for synthesising an adapter, which acts as an orchestrator and intervenes on the messages exchanged for correcting mismatch among peer interfaces. Most solutions existing for this problem assume peers interact synchronously via rendez-vous communication.

In this paper, we consider they exchange messages asynchronously via FIFO buffers. We also focus on a behavioural description model for peers, involving non-determinism and cycles. We propose new synthesis techniques for asynchronous communication semantics, which are based on an encoding into LNT, a modern process algebra. As far as adapter generation is concerned, we use the CADP toolbox for compiling the generated process algebraic specification to an LTS, and for minimizing the obtained result using classic reduction techniques. Beyond synthesis techniques, we also provide two kinds of verification. The first one relies on the stability property and aims at ensuring that the generated adapter will work from a certain size chosen for buffers. The second one is to use model checking techniques in order to verify that the adapter respect certain properties of interest. Our approach has been applied to several examples for validation purposes.

Our main perspective is to find some sufficient conditions on the LTS models or on the adaptation contract specification that could help ensuring the stability property preservation. Such conditions would avoid to check this property and ensure by construction that the generated adapter would work in unconstrained asynchronous environments.

References

1. L. Akroun, G. Salaün, and L. Ye. Automated Analysis of Asynchronously Communicating Systems. In *Proc. of SPIN'16*, volume 9641 of *LNCS*, pages 1–18. Springer, 2016.
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. P. Bertoli, M. Pistore, and P. Traverso. Automated composition of Web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3-4):316–361, 2010.
4. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.
5. 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.
6. J. Cámara, G. Salaün, C. Canal, and M. Ouederni. Interactive Specification and Verification of Behavioral Adaptation Contracts. *Information & Software Technology*, 54(7):701–723, 2012.
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. C. Canal and G. Salaün. Model-Based Adaptation of Software Communicating via FIFO Buffers. In *Proc. of FASE'15*, volume 9033 of *LNCS*, pages 252–266. Springer, 2015.
10. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 149 pages, 2011.
11. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.
12. F. Durán, M. Ouederni, and G. Salaün. A Generic Framework for N-Protocol Compatibility Checking. *Sci. Comput. Program.*, 77(7-8):870–886, 2012.
13. 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.
14. C. Gierds, A. J. Mooij, and K. Wolf. Reducing Adapter Synthesis to Controller Synthesis. *IEEE T. Services Computing*, 5(1):72–85, 2012.
15. 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.

16. J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour Analysis of Software Architectures*, pages 35–49. Kluwer Academic Publishers, 1999.
17. A. Marconi and M. Pistore. Synthesis and Composition of Web Services. In *Proc. of SFM'09*, volume 5569 of *LNCS*, pages 89–157. Springer, 2009.
18. 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.
19. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
20. 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.
21. M. Ouederni, G. Salaün, and T. Bultan. Compatibility Checking for Asynchronously Communicating Software. In *Proc. of FACS'13*, volume 8348 of *LNCS*, pages 310–328. Springer, 2013.
22. L. Padovani. Contract-Based Discovery and Adaptation of Web Services. In *Proc. of SFM'09*, volume 5569 of *LNCS*, pages 213–260. Springer, 2009.
23. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Trans. on Software Engineering*, 28(11):1056–1076, 2002.
24. 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.
25. 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.
26. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.