Model-Based Adaptation of Behavioral Mismatching Components

Carlos Canal, Pascal Poizat, *Member*, *IEEE Computer Society*, and Gwen Salaün

Abstract—Component-Based Software Engineering focuses on the reuse of existing software components. In practice, most components cannot be integrated directly into an application-to-be because they are incompatible. Software Adaptation aims at generating, as automatically as possible, adaptors to compensate for a mismatch between component interfaces and is therefore a promising solution for the development of a real market of components promoting software reuse. In this paper, we present our approach for software adaptation, which relies on an abstract notation based on synchronous vectors and transition systems for governing adaptation rules. Our proposal is supported by dedicated algorithms that automatically generate adaptor protocols. These algorithms have been implemented in a tool, called Adaptor, which can be used through a user-friendly graphical interface.

Index Terms—Software components, interfaces, mismatch, composition, software adaptation, adaptation contracts, vectors, transition systems, synchronous products, Petri nets, tools.

1 Introduction

Component-based Software Engineering (CBSE) aims at building new systems by assembling existing software components which would jointly realize the system desired functionality. However, one of the main issues raised by this paradigm is that, in practice, we cannot expect that any given software component perfectly matches the needs of a system where it is trying to be reused nor that the components being assembled perfectly fit one another. Reusing software often requires a certain degree of adaptation [1], [2], especially in the presence of legacy code. To deal with these problems, *Software Adaptation* [3], [4] is emerging as a new discipline concerned with providing techniques to arrange already developed pieces of software in order to reuse them in new systems, accommodating the potential mismatches arising from their composition.

Software Adaptation promotes the use of *adaptors*, specific computational entities developed for guaranteeing that a set of mismatching components will interact correctly. Software adaptation is different from software evolution, component customization, or adaptive middleware. *Software evolution* aims at modifying the code of the components, for instance to take a new functionality into account, whereas adaptation works in a nonintrusive way, that is, without modifying the code of the components, which is important due to their black-box nature. In the case of *customization*, the user may adjust the component behavior by tuning a fixed set of component parameters, which have been considered and defined at design time by the developer. Finally, dedicated *adaptive middleware* [5] can be used to put the adaptation process into action once an adaptor model has been obtained. In this sense, adaptive middleware complements software adaptation, which deals with adaptor modeling and synthesis, providing the means for the actual implementation of the proposal.

CBSE postulates that a component must be reusable from its interface [6], which in fact constitutes its full technical specification. The characteristics and expressiveness of the language used for interface description determines the degree of interoperability we can achieve using it and the kind of problems that can be solved. We distinguish several levels of interoperability and, accordingly, of interface description [2], [4], [7]: *technical* level (data encoding and framework-related issues), *signature* level (operation names and types), *behavioral* level (interaction *protocols*), *quality of service* level (nonfunctional properties such as security or efficiency), and *semantic* level, sometimes referred as conceptual level (functional specification of what the component actually does). At each one, mismatch may occur and have to be corrected. Industrial component models, by using Interface Description Languages (IDLs), are able to solve most of the technical interaction problems, but they fail to address mismatch at the higher levels. Numerous approaches have been presented for extending component interfaces with protocols (see, for instance, [8], [9], [10], [11], [12], [13]), thus resulting in what we call Behavioral IDLs (BIDLs). This interoperability level is essential because, even if components match from a signature point of view, their combination can lead to erroneous behaviors or deadlock situations if the designer is not aware of their execution flows and does not take them into account while building the full system.

- *C. Canal and G. Salaün are with LCC, Universidad de Málaga, Departamento Lenguajes y Ciencias de la Computacíon, ETSI Informática, Campus de Teatinos, 29701 Málaga, Spain. E-mail: {canal, salaun}@lcc.uma.es.*
- *P. Poizat is with IBISC FRE 3190 CNRS, Université d'Évry Val d'Essonne, Tour Evry 2, 523 place des terrasses de l'Agora, 91000 Evry, France, and ARLES Project-Team, INRIA, France. E-mail: pascal.poizat@inria.fr.*

Manuscript received 5 Mar. 2007; revised 1 Feb. 2008; accepted 15 Apr. 2008; published online 15 May 2008.
Recommended for acceptance by H. Schmidt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-03-0099.
Digital Object Identifier no. 10.1109/TSE.2008.31.

0098-5589/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

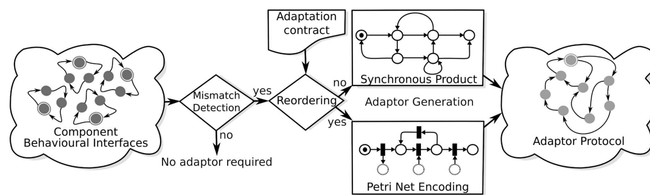

Fig. 1. Overview of our approach.

In this paper, we propose a model-based adaptation approach focusing on mismatch appearing at the behavioral level. Yet, since the component protocols are based on message exchange relative to the component operations, we also address name mismatch at the signature level. The approach (see Fig. 1 for a graphical overview of it) takes as input the behavioral interfaces of components to be adapted and an adaptation *contract* [4], which is an abstract description of the constraints which must be respected to make the involved components work together. Given these two elements, an adaptor protocol is generated in an automatic way.

The adaptation process begins with two (or more) components that are not able—as they are—to interact successfully (i.e., ending in correct termination states). To compensate such mismatch, we propose using synchronous vectors as adaptation contract language to make explicit the interactions between components, possibly on different message names. Our notation also allows the specification of ordering constraints on interactions, which enables one to describe in an abstract way more complex adaptation scenarios. In order to generate adaptor protocols for such contracts, we present in this paper two algorithms that automate the adaptation process. The first is based on synchronous products and the second is based on Petri net encodings. Compared to the former, the latter induces a higher computational complexity, but is able to reorder messages when necessary and then ensures a correct interaction when several components have the messages exchanged in their protocols which are not ordered correspondingly. Reordering is worked out by desynchronizing the message emission by one component and the message reception in another one. When required, emitted messages are temporarily memorized until they are used for effective interaction. This is why a formalism capable of representing memory, such as Petri nets, is required. The adaptation techniques we present in this paper have been implemented in a tool, called Adaptor, which has been applied to many nontrivial examples, i.e., examples where adaptor protocols could not have been obtained by hand.

A preliminary version of this work has been presented in [14] and is extended here in several aspects:

1. introduction to the Petri nets concepts used in our proposal,
2. detailed descriptions of the adaptation algorithms,
3. presentation of the Adaptor tool,
4. illustration on a more realistic and bigger case study from the pervasive computing domain, and
5. an updated review and comparison with related work.

The remainder of the paper is organized as follows: Section 2 formally introduces our component interface model and defines interface mismatch. Section 3 focuses on the adaptation contract notation. Section 4 presents a first approach to component adaptation based on synchronous products. Section 5 presents a second solution which goes further, considering reordering through the encoding of contracts and behavioral interfaces into Petri nets. Section 6 gives an overview of the Adaptor tool. In Section 7, we survey the more advanced proposals for behavioral software adaptation and compare to them. Finally, Section 8 ends the paper with some concluding remarks. Proofs are given in [15].

2 Interfaces and Mismatch

In this section, we first present the model of interfaces through which components are accessed and used. Then, we define the notion of interface mismatch that our approach addresses.

2.1 Component Interfaces

We assume that component interfaces are given using both a signature and a behavioral interface. Signature interfaces usually correspond in component-based frameworks (e.g., CCM, .NET, or J2EE) to operation profiles described using an IDL, i.e., operation names associated with argument and return types relative to the data being exchanged when the operation is called. Since we focus on the behavioral level in this paper, we omit the elements relative to data exchange in the signature interfaces. This means that a signature is taken as a disjoint set of provided and required operation names. Such abstractions from data exchange are often used in software engineering, e.g., to check interface compatibility [11] or to perform component verification [10], [12]. Additionally, we propose that behavioral interfaces are represented by means of Labeled Transition Systems (LTSs). Message-based communication between components is therefore represented using *events* relative to the emission (denoted using !) and reception (denoted using ?) of *messages* corresponding to operation calls.

However, taking data exchange into account is important to ensure full compatibility. So far, this can be supported in our approach using additional messages as follows: The emission by a component of a message login with two data information, username and password, would be encoded by the sequence of events login!.username!.password! in the component LTS. Accordingly, the reception in a component of a message login with two data information, username and password, would be encoded by the sequence of events login?.username?.password? in the component LTS. Provided this encoding is performed as preprocessing and the adaptation contract takes the additional messages into account, the protocols can be adapted, as demonstrated in [16] where we have applied our adaptation techniques to Windows Workflow Foundation (WF) [17], which belongs to the .NET Framework 3.0 developed by Microsoft. Related perspectives are further discussed in Section 8.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

Definition 1 (LTS). *A* Labeled Transition System *is a tuple* (A, S, I, F, T) *where* A *is an alphabet (set of events),* S *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are final states, and* $T \subseteq S \times A \times S$ *is the transition function.*

Final states correspond to correct service terminations in components. To support the correctness of the adaptation process, we further assume that the initial state is also final ($I \in F$). The alphabet of the LTS is built on the component signature. This means that, for each provided operation p in the signature, there is a message p and an event $p?$ in the alphabet and, for each required operation r, there is a message r and an event $r!$ in the alphabet. Complementary events are denoted with the same name of message and opposite directions. Consequently, the complementing function on events is defined as: $\overline{e?} = e!$, and $\overline{e!} = e?$.

LTSs are adequate models as far as user-friendliness and development of formal algorithms are concerned. However, higher-level languages such as process algebras [18] can be used to define behavioral interfaces in a more concise way. In a former version of this work [14], the sequential subset of CCS [19] was used as BIDL. Moreover, CCS descriptions of component behavioral interfaces can be easily translated into LTS models using the operational rules defining the semantics of the formalism. In this paper, since we focus on the adaptor model generation, we only present and work using LTS models. In [16], the reader will find more details of how LTSs can be extracted from component languages (namely, in this work, the Windows WF language) and how an adaptor model can be transformed into a component language program.

Example 1. eMuseum is an added-value application whose objective is to augment the visitors' experience in museums by displaying, on their portable devices, information about seen pieces of art. We will use this example throughout the paper. Let us first begin with a simplified version of it. eMuseum is built using two separately designed components: a room server (ROOM) and a Personal Digital Assistant application (PDA). On the one hand, ROOM can be asked (query message) to send a list of artifacts present in the room (list message) and is then informed about one being selected (choice message). ROOM may afterward be requested to send information about this artifact either in textual (pdf) or video (mpeg) format using, respectively, the textrequest and videorequest messages. The files themselves are sent with the text or the video message. On the other hand, PDA first issues a resource discovery query, then may be used to select a given item from a list of available resources, and the resource is eventually displayed (mpeg or pdf). PDA can also be turned off using the shutdown message. The LTSs of the components are given in Fig. 2, with initial and final states, respectively, marked using bullet arrows (e.g., state 0 in PDA) and hollow states (e.g., states 0 and 4 in PDA). Transitions sharing the same source and target states are represented using a single transition and the list of the possible labels.

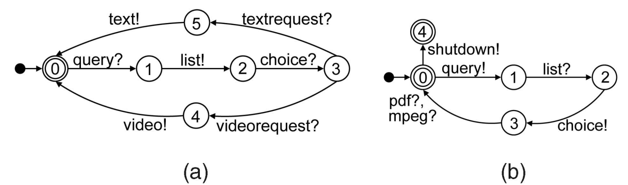

Fig. 2. eMuseum, version 1. (a) ROOM LTS. (b) PDA LTS.

2.2 Behavioral Mismatch

Mismatch situations between component interfaces may be caused by message names that do not correspond, by an ordering of messages which is not compatible in two or more components, or by some messages in one component that have no counterpart or match with several messages in another component (one-to-zero, one-to-many, or many-to-one correspondences). All of these cases of behavioral mismatch can be worked out using the contract notation (Section 3) and the adaptation algorithms (Sections 4 and 5) that we propose in this paper. We will give examples of such mismatch in the case study we present in the sequel.

There exist numerous definitions of compatibility and, as a consequence, of mismatch between protocols [4], [20], but deadlock is the most commonly accepted notion. To automate deadlock mismatch detection, the first step is to define the semantics of a system composed of several components. This semantics can be given by means of the synchronous product [21] of LTSs. The synchronous product of several component LTSs results in a new LTS which contains all of the possible interactions between the involved components, assuming they synchronize on complementary events $(a, \overline{a})$.

Definition 2 (Synchronous Product). *The* synchronous product *of* n *LTSs* $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \dots, n\}$, *is the LTS* $C_1 \| \dots \| C_n = (A, S, I, F, T)$ *such that*

- $A = A_1 \cup \{_\} \times \dots \times A_n \cup \{_\}$, $S = S_1 \times \dots \times S_n$, $I = (I_1, \dots, I_n)$, $F = F_1 \times \dots \times F_n$ *and*
- T *is defined using the following rule:* $\forall (s_1, \dots, s_n) \in S$, $\forall i, j \in \{1, \dots, n\}$, $i < j$, *such that* $\exists (s_i, a, s'_i) \in T_i$, $\exists (s_j, \overline{a}, s'_j) \in T_j$: $(x_1, \dots, x_n) \in S$ *and* $((s_1, \dots, s_n), (l_1, \dots, l_n), (x_1, \dots, x_n)) \in T$, *where* $\forall k \in \{1, \dots, n\}$:

$$\begin{cases} l_k = a, x_k = s'_i & \text{if } k = i, \\ l_k = \overline{a}, x_k = s'_j & \text{if } k = j, \\ l_k = _, x_k = s_k & \text{otherwise,} \end{cases}$$

where the $\times$ *operator stands for the Cartesian product.*

The states in the product correspond to sets of states of the components (called substates in the context of a product state). For example, a state $(s_1, \dots, s_n)$ denotes that each component C_i is its state s_i. Initially, all components are in their initial state (i.e., I_i for each C_i), which means that the initial state of the product is $(I_1, \dots, I_n)$. The computation of the transitions expresses that, given some composite state $(s_1, \dots, s_n)$ in the product, there is some transition outgoing from this state iff there are two components, i and j, that may perform at the same time—from states s_i and s_j in

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

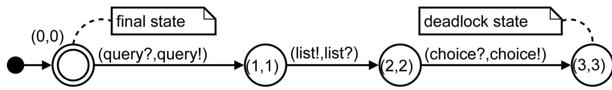

Fig. 3. Synchronous product for Example 1 LTSs.

their LTS—complementary events (i.e., one sending a message and the other one receiving it), while other components do not perform any action (denoted _). The resulting target state of the product transition corresponds to the source state of it, but for the substates corresponding to components i and j. Transitions in the product are labeled with a set of labels, one from each component (including _). An example of synchronous product is given in Example 2.

We are now able to characterize mismatch by means of an adequate definition of deadlock that differentiates deadlock states and correct final states. A system is blocked when it cannot evolve and when at least one of the components is not in one of its final states.

Definition 3 (Deadlock State). *Let $C = (A, S, I, F, T)$ be an LTS. A state s is a deadlock state for C, noted $dead(s)$, iff it is in S, not in F, and has no outgoing transitions: $s \in S \wedge s \notin F \wedge \nexists l \in A,\ s' \in S\ .\ (s, l, s') \in T$.*

Definition 4 (Deadlock Mismatch). *An LTS $C = (A, S, I, F, T)$ presents a deadlock mismatch if there is a state s in S such that $dead(s)$.*

To check if a system composed of several components presents mismatch, its synchronous product is computed and then Definition 4 is used. Synchronous products and deadlock detection are common in the Formal Methods community and, hence, are supported by tools such as CADP [22], a toolbox dedicated to the validation and verification of concurrent systems. However, our deadlock definition is slightly different from the one used in these tools since it has to distinguish between success (deadlock in a final state) and failure (deadlock in a nonfinal state). Yet, behavioral mismatch detection can be automatically checked, e.g., by CADP, up to the adding within component interfaces of loop transitions over final states labeled with a specific label (we use accept).

Example 2. In the synchronous product of the ROOM and the PDA components (Fig. 3), a deadlock state, (3, 3), is reached after three successful interactions as this state 1) has no output transitions and 2) is not final. The latter is caused by the fact that the corresponding states in the ROOM (state 3) and PDA (state 3) components are not final, while both should be for (3, 3) to be final. The former is caused by the name mismatch between, respectively, the PDA messages mpeg and pdf and the ROOM messages textrequest and videorequest. One would also note that the shutdown message in PDA has no counterpart in ROOM. Hence, there is no possible sequence of transitions leading to the other potential final state in the product, i.e., state (0, 4), corresponding to state 0 of ROOM and state 4 of PDA.

3 Adaptation Contracts

In this section, we present the adaptation contract notation that enables one to specify how to work out mismatch situations. We rely on *synchronous vectors* [21], which denote communication between several components, where each event appearing in one vector is executed by one component and the overall result corresponds to a synchronization between all of the involved components. A vector may involve any number of components and does not require interactions to occur on the same names of events. Vectors can describe expressive communication patterns, which is especially useful to express n-ary interactions.

Definition 5 (Vector). *A* synchronous vector *(or* vector *for short) for a set of components $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \dots, n\}$, is a tuple $\langle e_1, \dots, e_n \rangle$ with $e_i \in A_i \cup \{_\}$, _ meaning that a component does not participate in a synchronization.*

In order to unambiguously identify every communication in the adaptor, prior to the adaptation process, component event names are prefixed by the component name, e.g., PDA:query! or ROOM:query?. Yet, to favor readability, prefixes are not given in component LTS when they are clear from the context.

Example 3. Let us get back to the eMuseum example. We first define vectors for messages that match:

$$\begin{aligned} \mathrm{v_{query}} &= \langle \mathrm{ROOM : query?, PDA : query!} \rangle, \\ \mathrm{v_{list}} &= \langle \mathrm{ROOM : list!, PDA : list?} \rangle, \text{ and} \\ \mathrm{v_{choice}} &= \langle \mathrm{ROOM : choice?, PDA : choice!} \rangle. \end{aligned}$$

Further, we have seen that mismatch came first from the unanticipated shutdown reception. This would be solved by a specific vector, $\mathrm{v_{end}} = \langle \mathrm{ROOM : _, PDA : shutdown!} \rangle$, to specify that the adaptor should not transmit the shutdown message to the ROOM server. Moreover, mismatch also came from the text/video choice (using textrequest or videorequest) which is not done by PDA, which waits for one resource to be sent, either with the pdf or the mpeg message. A possible solution would require to express that the video (respectively, text) choice is performed by the adaptation itself using vectors $\mathrm{v_{vmode}} = \langle \mathrm{ROOM : videorequest?, PDA : _} \rangle$ and $\mathrm{v_{tmode}} = \langle \mathrm{ROOM : textrequest?, PDA : _} \rangle$. Moreover, we would like to specify a correspondence between the video sending (video in ROOM) and the mpeg file reception (mpeg in PDA) and a correspondence between the text sending (text in ROOM) and the pdf file reception (pdf in PDA). The corresponding vectors would be $\mathrm{v_{vget}} = \langle \mathrm{ROOM : video!, PDA : mpeg?} \rangle$ and $\mathrm{v_{tget}} = \langle \mathrm{ROOM : text!, PDA : pdf?} \rangle$.

Vectors express correspondences between messages, like bindings between ports or connectors in architectural descriptions [23]. Yet, vectors alone are not sufficient to perform adaptation as one must also take into account the context in which messages are exchanged, i.e., the component protocols. Suppose we have a vector $\langle \mathrm{c_1 : a!, c_2 : b?} \rangle$. Directly sending, in an adaptor, message b

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

to c_2 when message a is received from c_1 may lead the system to a deadlock state if this interaction is incorrect. This is why more complex adaptation algorithms such as the ones we define in this paper are required. Moreover, vectors are not sufficient to support more advanced adaptation scenarios such as contextual rules, choice between vectors, or, more generally, ordering (e.g., when one message in some component corresponds to several in another component, which requires applying several vectors). The ordering in which vectors have to be applied can be specified using different notations such as regular expressions, LTSs, or (Hierarchical) Message Sequence Charts. Due to their readability and user-friendliness, we chose to specify adaptation contracts using *vector LTSs*, that is, LTSs whose labels are vectors. In addition, vector LTSs facilitate the development of adaptation algorithms since they provide an explicit description of the contract behaviors set of states, which makes their traversal easier. Other notations, such as the ones mentioned above, can be used to specify the adaptation contract provided that they can be translated into vector LTSs. To this purpose, one can rely on existing behavioral model synthesis techniques such as those presented in [24] for regular expressions or in [25] for Message Sequence Charts.

Definition 6 (Vector LTS). *A* vector LTS *for a set of vectors V is an LTS (V, S, I, F, T), where labels are vectors.*

Definition 7 (Adaptation Contract). *An* adaptation contract *for a set of components $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \ldots, n\}$, is a couple (V, L), where V is a set of vectors for components C_i and L is a vector LTS for V.*

If only message name correspondences are necessary to solve mismatch between components, the vector LTS may leave the vector application order unconstrained, using a single state and all vector transitions looping on it. In particular, this pattern may be used on specific parts of the contract for which the designer does not want to impose any ordering.

The design of the adaptation contracts is the only step of adaptation which is not handled automatically by our approach. Yet, this step is essential because an inadequate contract could induce the generation of an adaptor that would ensure deadlock freedom at the cost of too many interaction removals, including ones expected by the designer. Solutions and on-going work relative to contract design are discussed in Section 8.

Example 4. Using the vectors given in Example 3, one could express different adaptation contracts (Fig. 4). A simple example is contract 1. This contract is limited to video exchange as it does not use vectors for text exchange (v_{tmode} and v_{tget}). Except for this, the contract is very permissive. It enables any application ordering of name mismatch resolution using the vectors, including when no video is ever exchanged (i.e., vectors v_{vmode} and v_{vget} may never be applied). One could have either text or video be exchanged with contract 2. Here, at each PDA request, the adaptor willy be able to choose nondeterministicall between text and video. One could also enforce a very strict adaptation contract with contract 3, where textual and video information are alternatively used. Note that the use of such highly constrained contracts, applied to adaptation without reordering, is not very interesting as giving such a contract is often close to giving the solution, while using more permissive contracts and adaptation with reordering demonstrates the full power of our automated adaptation process. Other contracts will be presented in the sequel, together with the different algorithms that operate on them to produce the corresponding adaptor protocols.

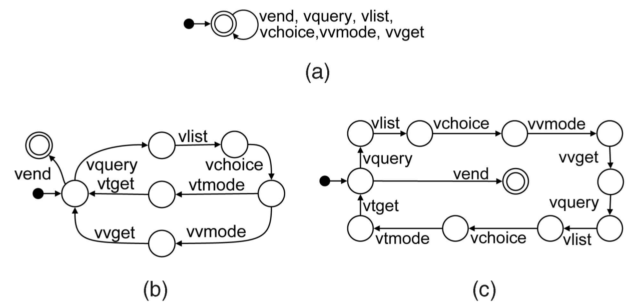

Fig. 4. Adaptation contracts for eMuseum, version 1. (a) Contract 1. (b) Contract 2. (c) Contract 3.

An adaptor is given by an LTS which, put into a nondeadlock-free system, yields a deadlock-free one. All of the exchanged messages will pass through the adaptor, which can be seen as a coordinator for the components to be adapted. This can be formalized as follows:

Definition 8 (Adaptation Algorithm Correctness). *Given n components C_i, $i \in \{1, \ldots, n\}$, and a contract, the adaptation algorithm builds an adaptor Ad such that there is no deadlock state in the system $Ad \| (C_1 \| \ldots \| C_n)$.*

In the sequel, we present two different correct algorithms for the generation of adaptor protocols.

4 Adaptation without Reordering

In this section, we present a first adaptation algorithm, based on synchronous products. More precisely, we rely on an extension of the synchronous product, Definition 2, that takes into account not only the correspondences of events described in the vectors but also their ordering in the vector LTS. Consequently, the vector LTS is used as a guide to build the resulting product.

Definition 9 (Synchronous Vector Product). *The* synchronous vector product (with vector LTS) *of n LTS $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, \ldots, n\}$, with a vector LTS $L = (A_L, S_L, I_L, F_L, T_L)$, is the LTS $\Pi_L((C_1, \ldots, C_n), L) = (A, S, I, F, T)$ such that*

- $A = A_L \times A_1 \cup \{_\} \times \ldots \times A_n \cup \{_\}$, $S = S_L \times S_1 \times \ldots \times S_n$, $I = (I_L, I_1, \ldots, I_n)$, $F = F_L \times F_1 \times \ldots \times F_n$ *and*
- T *contains a transition* $((s_L, s_1, \ldots, s_n), \langle a_L, a_1, \ldots, a_n \rangle, (s'_L, s'_1, \ldots, s'_n))$ *iff there is a state* $(s_L, s_1, \ldots, s_n)$ *in* S, *there is a transition*

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

$(s_L, \langle l_1, \dots, l_n \rangle, s'_L)$ in T_L, and, for every i in $\{1, \dots, n\}$:

- *if $l_i = _$, then $s'_i = s_i$ and $a_i = _$;*
- *otherwise, there is a transition (s_i, a_i, s'_i) with $a_i = l_i$ in T_i.*

As with Definition 2, states in the product correspond to sets of states of the components, but also take into account the vector LTS. For example, a state $(s_0, s_1, \dots, s_n)$ denotes that each component C_i is in its state s_i and that the vector LTS is in s_0. Initially, all components and the vector LTS are in their initial state (i.e., I_i for each C_i and I_L for the vector LTS), which means that the initial state of the product is $(I_L, I_1, \dots, I_n)$. The computation of the transitions is also slightly different from Definition 2. There is an outgoing transition from a state $(s_L, s_1, \dots, s_n)$ iff there is a transition labeled by a vector $\langle l_1, \dots, l_n \rangle$ outgoing from state s_L in the vector LTS and, as a consequence, if, for every component C_i, there is a transition outgoing from s_i and labeled with l_i in the C_i LTS. A commented example of synchronous vector product computation is given in Example 5 (Fig. 7).

To generate an adaptor protocol from a synchronous vector product, we have to discard the first element of the product components to keep only the elements corresponding to the component exchanges. More formally, it means that, from an LTS $P_L = \Pi_L((C_1, \dots, C_n), L) = (A, S, I, F, T)$, we compute the LTS $P = proj(P_L) = (A', S', I', F', T')$ such that $\forall X \in \{A, S, I, F\}$,

$$X' = \{cdr(x) \mid x \in X\}$$

and

$$T' = \{(cdr(s), cdr(l), cdr(s')) \mid (s, l, s') \in T\}$$

with $cdr((x_0, x_1, \dots, x_n)) = (x_1, \dots, x_n)$.

Our algorithm (Algorithm 1) takes as input a set of component LTSs C_i and an adaptation contract (V, L). This algorithm is based on three main steps: 1) computation of the synchronous vector product taking the vector LTS L into account and discarding in the result the vector LTS elements (Algorithm 1:1-2), 2) removal of interaction sequences (paths) leading to deadlock (function *remove_-deadlocks*, Algorithm 1:3), and 3) for each transition (Algorithm 1:5-14), reversal of the directions for all events appearing in the vector on the transition, called mirroring (Algorithm 1:6-7), and computation of all possible interleavings (function *compute_permutations*) starting with receptions (Algorithm 1:8-13).

Algorithm 1 *build_adaptor_noreordo*
constructs an adaptor without reordering for a set of components given an adaptation contract
inputs components $C_1, \dots, C_n$ with each
$C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, and an adaptation contract (V, L)
output adaptor $Ad = \langle A, S, I, F, T \rangle$
1: $P_L := \Pi_L((C_1, \dots, C_n), L)$
2: $P = proj(P_L)$
3: $P_{restr} := remove_deadlocks(P)$
4: $S_{\text{add}} := \emptyset$; $T_{Ad} := \emptyset$
5: **for all** $t = (s = (s_1, \dots, s_n), (l_1, \dots, l_n), s' = (s'_1, \dots, s'_n))$ in $T_{P_{\text{restr}}}$ **do**
6: $L_{\text{rec}} = \{l? \mid l! \in (l_1, \dots, l_n)\}$
7: $L_{\text{em}} = \{l! \mid l? \in (l_1, \dots, l_n)\}$
8: $\text{Seq}_{\text{rec}} = compute_permutations(L_{\text{rec}})$
9: $\text{Seq}_{\text{em}} = compute_permutations(L_{\text{em}})$
10: **for all** $(R = (r_1, \dots, r_i), E = (e_1, \dots, e_p)) \in \text{Seq}_{\text{rec}} \times \text{Seq}_{\text{em}}$ **do**
11: $T_{Ad} := T_{Ad} \cup \{s \xrightarrow{r_1} q_1, \dots, q_{i-1} \xrightarrow{r_i} q_i, \dots, q_{i+1} \xrightarrow{e_1} q_{i+2}, \dots, q_{n-1} \xrightarrow{e_p} s'\}$
12: $S_{\text{add}} := S_{\text{add}} \cup \{q_1, \dots, q_{n-1}\}$
13: **end for**
14: **end for**
15: **return** $Ad = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}} \cup S_{\text{add}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_{Ad})$

Removing deadlock paths [26] is required to suppress spurious interactions that would not leave the system in a stable (final) state, as shown in Example 5. This is achieved by recursively removing transitions and states yielding deadlocks: Find a state s such that $dead(s)$, remove s and any transition t with target s, and do this until there is no more such s in the LTS. Mirroring ensures that the adaptor and the components can perfectly communicate using the same event message names with opposite directions (!/? or ?/!). Moreover, event interleaving is essential when vectors involve more than two events in a communication (e.g., in case of broadcast or multicast communication). Interleavings make the adaptor support nondeterminism *wrt* the orderings in which events will occur, hence accepting any possible one.

Note that Algorithm 1 builds an adaptor protocol by applying one vector after the other, that is, all interactions involved in one vector occur before starting the interactions of another vector. Consequently, events belonging to two vectors appearing as labels in the synchronous product are never interleaved. Such an interleaving is mandatory when events need to be reordered. This additional feature will be supported by the algorithm presented in Section 5. The complexity of Algorithm 1 lies in the synchronous vector product computation and is $\mathbf{O}(|S|^{n+1})$, where S is the largest set of states for all component (and vector) LTS and $n+1$ stands for the n components plus the vector LTS.

Example 5. Let us now present a second version of eMuseum. A new version of the ROOM component supports an additional feature: Once a video has been sent, it can be resent (upon reception of the again message) to be played again. The quit message is then used to tell ROOM one is done with the selected video. The ROOM designer has also refactored this component. The names of some operations (namely, query and choice) and, as a consequence, of the corresponding messages, have been changed. A new version of the PDA component is also used. It now supports being integrated in contexts where rights can be different depending on two modes: a guest mode (with fewer rights) and a user mode (with more rights). PDA can send login (respectively, logout) messages to go from guest to user mode (respectively, from user to guest mode). The new interfaces of the two components are given in Fig. 5 (changes are in bold).

As far as the adaptation contract is concerned, one does not start from scratch. The vectors we had before

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

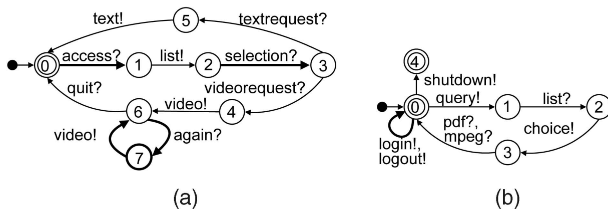

Fig. 5. eMuseum, version 2. (a) ROOM LTS. (b) PDA LTS.

are reused, replacing old messages by new ones where we have now name mismatch (in bold font):

$$\begin{aligned}&v_{\text{end}} = \langle \text{ROOM} : _, \text{PDA} : \text{shutdown!} \rangle,\\ &v_{\text{vmode}} = \langle \text{ROOM} : \text{videorequest?}, \text{PDA} : _ \rangle,\\ &v_{\text{vget}} = \langle \text{ROOM} : \text{video!}, \text{PDA} : \text{mpeg?} \rangle,\\ &v_{\text{tmode}} = \langle \text{ROOM} : \text{textrequest?}, \text{PDA} : _ \rangle,\\ &v_{\text{tget}} = \langle \text{ROOM} : \text{text!}, \text{PDA} : \text{pdf?} \rangle,\\ &v_{\text{query}} = \langle \mathbf{ROOM : access?}, \text{PDA} : \text{query!} \rangle,\\ &v_{\text{list}} = \langle \text{ROOM} : \text{list!}, \text{PDA} : \text{list?} \rangle, \text{ and}\\ &v_{\text{choice}} = \langle \mathbf{ROOM : selection?}, \text{PDA} : \text{choice!} \rangle.\end{aligned}$$

We also add vectors for unspecified receptions of messages sent by the PDA for changing mode as ROOM has not been built *wrt* such modes: $\mathbf{v_{user}} = \langle \mathbf{ROOM :_}, \mathbf{PDA : login!} \rangle$ and $\mathbf{v_{guest}} = \langle \mathbf{ROOM :_}, \mathbf{PDA : logout!} \rangle$. The support for changing mode and, more generally, contexts will be achieved using the vector LTS, below. Finally, we add vectors corresponding to the new feature of ROOM (resending videos): $\mathbf{v_{again}} = \langle \mathbf{ROOM : again?}, \mathbf{PDA :_} \rangle$ and $\mathbf{v_{quit}} = \langle \mathbf{ROOM : quit?}, \mathbf{PDA :_} \rangle$. The adaptor will be in charge of sending them when required, as for the video and text requests. Note that if we had used a single vector $\langle \text{ROOM} : \text{quit?}, \text{PDA} : \text{shutdown!} \rangle$ in place of v_{end} and v_{again}, we would have enforced that ROOM and PDA exchange information exactly once (forbidding the PDA to shut down directly and to ask several times for information).

By using a vector LTS (Fig. 6), we will enforce the following constraints:

- There are two modes, GUEST and USER. In the eMuseum application, we take the benefit of these two modes as follows: In GUEST mode, the sent information is text. In USER mode, the sent information is video. This demonstrates how an adaptation contract can be used to enforce constraints which are defined system-wide, not at the level of individual components.
- The two modes alternate (starting in GUEST mode), going from one to another using the login and logout messages.
- We know that communication is based on two phases, selection and getting information, yet we keep an abstract description level for these. Nondeterminism may be kept in the contract, e.g., in USER mode, between different possible application orderings of v_{query}, v_{list}, v_{choice}, v_{vmode}, v_{vget}, v_{again}, and v_{vquit} to let the adaptation process decide which one—if any—is correct (see the corresponding adaptor, Fig. 8). For this, the adaptation process uses the orderings which are defined in the component interfaces.

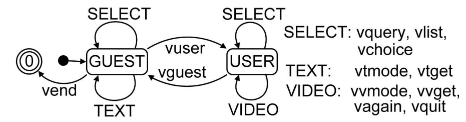

Fig. 6. Adaptation contract for eMuseum, version 2.

In order to generate the adaptor protocol, we first compute the synchronous vector product (Fig. 7) of the ROOM LTS (Fig. 5a) and the PDA LTS (Fig. 5b) with the vector LTS (Fig. 6). To understand how this works, let us take, for example, the computation of the transitions outgoing from the product initial state. This initial state, (0, 0, GUEST), corresponds to the composition of the components' and vector's LTS initial states. Different sets of transitions are possible in the three LTSs used in the product:

- access? in ROOM;
- shutdown!, login!, logout!, and query! in PDA; and
- vectors

$$\begin{aligned}&v_{\text{end}}(\langle \text{ROOM} : _, \text{PDA} : \text{shutdown!} \rangle),\\ &v_{\text{user}}(\langle \text{ROOM} : _, \text{PDA} : \text{login!} \rangle),\\ &v_{\text{query}}(\langle \text{ROOM} : \text{access?}, \text{PDA} : \text{query!} \rangle),\\ &v_{\text{list}}(\langle \text{ROOM} : \text{list!}, \text{PDA} : \text{list?} \rangle),\\ &v_{\text{choice}}(\langle \text{ROOM} : \text{selection?}, \text{PDA} : \text{choice!} \rangle),\\ &v_{\text{tmode}}(\langle \text{ROOM} : \text{textrequest?}, \text{PDA} : _ \rangle), \text{ and}\\ &v_{\text{tget}}(\langle \text{ROOM} : \text{text!}, \text{PDA} : \text{pdf?} \rangle)\end{aligned}$$

  in the vector LTS.

Therefore, there are only three possible transitions outgoing from the product initial state (corresponding to the first three vectors above):

- $$\langle \langle \text{ROOM} : _, \text{PDA} : \text{shutdown!} \rangle, \text{ROOM} : _, \text{PDA} : \text{shutdown!} \rangle,$$
  going to state (0, 4,0);
- $$\langle \langle \text{ROOM} : _, \text{PDA} : \text{login!} \rangle, \text{ROOM} : _, \text{PDA} : \text{login!} \rangle,$$
  going to state (0,0,USER); and
- $$\langle \langle \text{ROOM} : \text{access?}, \text{PDA} : \text{query!} \rangle, \text{ROOM} : \text{access?}, \text{PDA} : \text{query!} \rangle,$$
  going to state (1,1,GUEST).

The other possibilities are forbidden, either because one component corresponding to a message in a possible vector is not ready for it (e.g., ROOM cannot receive textrequest in its initial state, 0) or because components may be ready for some message but the contract forbids it (e.g., PDA may send logout but vector v_{guest} is not enabled in the initial state of the vector LTS, (0, 0, GUEST)). We may proceed similarly, step by step,

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

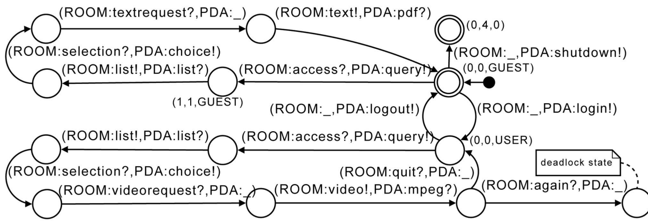

Fig. 7. Product LTS for eMuseum, version 1.

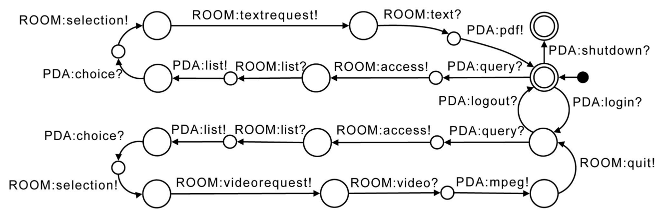

Fig. 8. Adaptor protocol for eMuseum, version 2.

now computing, for example, the transitions outgoing from the (0, 4, 0), (0, 0, USER), and (1, 1, GUEST) states. The result is given in Fig. 7, where the parts of the labels corresponding to the vectors are discarded due to place matters (i.e., *wrt* Algorithm 1, we give P in place of P_L).

One may note a path leading to a deadlock state in this example. After ROOM and PDA have successfully exchanged a first video, the adaptor may have ROOM send it again using the again message. However, ROOM would then send the video message, which would block the system as PDA is not ready to receive the corresponding mpeg message. Indeed, this could have been prevented by removing vector $\mathrm{v}_{\mathrm{again}}$ from the adaptation contract. Yet, as one cannot ensure that the perfect contract is always given, it shows the need for the suppression of spurious interactions after the product is computed.

We finally compute the adaptor by mirroring the labels and computing permutations of inputs and then outputs for it (see Fig. 8). In this adaptor protocol, we see that behavioral mismatch (one-to-zero, i.e., unanticipated reception, such as shutdown, one-to-one such as query versus access, many-to-one such as videorequest and video versus mpeg) have been worked out. The adaptor follows a coordination protocol which is restricted to the contract and modes we specified (e.g., text is sent to guests and video to users). Finally, the adaptor has also removed all possible interaction sequences leading to deadlocks (as demonstrated above with video resending).

5 Adaptation with Reordering

Let us now extend the domain of mismatch problems we deal with. Our goal is now to also address behavioral mismatch which requires reordering. This occurs when exchanged messages present noncompatible orderings in the components' protocols. To support this kind of mismatch, the adaptation process may try to accommodate protocols by reordering events in between the components. The behavioral adaptation proposal presented in Section 4 may yield an empty adaptor in the presence of such a mismatch because it induces application of one vector after the other and, therefore, prevents the application of several vectors at the same time, which is necessary to make reordering effective.

To this purpose, we present a second approach which complements the one presented in Section 4. Messages received by the adaptor are seen as *resources* which are memorized until they need to be sent (i.e., until they may be received by some component to make it evolve). This can be achieved first due to an *encoding* of the component protocols and of the adaptation contract into a formalism that supports a *memory* and a *resource-based vision of adaptation*, as follows: 1) reception of messages (by the adaptor) corresponds to a resource creation, 2) emission of messages (by the adaptor) is possible provided some resource is available and corresponds to resource consumption, and, finally, 3) vectors correspond to resource transfer. Petri nets [27] are such a formalism, which further benefits from good tool support. Moreover, the marking graph of such a Petri net encoding represents all possible resource-based evolutions of the adaptor (message reception, emission, and transfer).

Before presenting our algorithm for adaptation with reordering in more details, let us first introduce the basics of Petri nets. A Petri net consists of places, transitions, and directed arcs between places and transitions. A transition is connected by input arcs to a set of input places and by output arcs to a set of output places. Places may contain any number of tokens that model resources. Transitions act on tokens by a process known as *firing*. A transition can be fired if there are enough tokens in each of its input places. When a transition fires, it consumes one token from each of its input places, and adds a token into each of its output places. The presentation of Petri nets is simplified here for conciseness purposes as, e.g., generalized Petri nets support arcs labeled with natural numbers to denote the need of more than one token in an input place and the production of more than one token in an output place. A distribution of tokens over the places of a net is called a *marking*. A *marking graph* describes all of the markings that can be reached from an initial marking by firing transitions.

Algorithm 2 takes as input a set of component LTSs C_i and an adaptation contract and generates the corresponding Petri net encoding.

Algorithm 2 *build_PetriNet*
constructs a Petri net encoding from component interfaces and an adaptation contract
inputs components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, and an adaptation contract $(V, L = (A_L, S_L, I_L, F_L, T_L))$
output Petri net $\mathcal{N}$
1: $\mathcal{N} := empty_PetriNet()$ // *the following operates on* $\mathcal{N}$
2: **for all** $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$, $i \in \{1, \ldots, n\}$ **do**
3: **for all** $s_j \in S_i$ **do** add a place $C_i\!:\!s_j$ **endfor**
4: put a token in place $C_i\!:\!I_i$ // I_i: *initial state of* C_i
5: **for all** $a! \in A_i$ **do** add a place $??C_i\!:\!a$ **endfor**
6: **for all** $a? \in A_i$ **do** add a place $!!C_i\!:\!a$ **endfor**
7: **for all** $(s, e, s') \in T_i$ with $l = e$ **do**

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

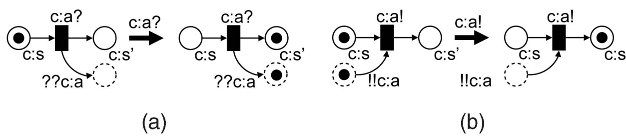

Fig. 9. Encoding patterns for component protocols (and related marking evolution semantics). (a) Message emission in components/reception in adaptor. (b) Message reception in components/emission in adaptor.

8: add a transition with label $\bar{l}$, an arc from place $C_i : s$ to the transition and an arc from the transition to place $C_i : s'$
9: **if** l has the form $a!$ **then** add an arc from the transition to place $??C_i : a$ **endif**
10: **if** l has the form $a?$ **then** add an arc from place $!!C_i : a$ to the transition **endif**
11: **end for**
12: **end for**
13: **for all** $s_L \in S_L$ **do** add a place $C_L : s_L$ **endfor**
14: put a token in place $C_L : I_L$ // I_L: *initial state of* L
15: **for all** $t_L = (s_L, \langle e_1, \ldots, e_n \rangle, s'_L) \in T_L$ with $\forall i \in \{1, \ldots, n\} l_i = e_i$ **do**
16: add a transition with label tau, an arc from place $C_L : s_L$ to the transition and an arc from the transition to place $C_L : s'_L$
17: **for all** l_i **do**
18: **if** l_i has the form $a!$ **then** add an arc from place $??C_i : a$ to the transition **endif**
19: **if** l_i has the form $a?$ **then** add an arc from the transition to place $!!C_i : a$ **endif**
20: **end for**
21: **end for**
22: **for all** $(f_r, f_1, \ldots, f_n) \in F_L \times F_1 \times \ldots \times F_n$ **do**
23: add a (loop) accept transition with arcs from and to each of the tuple elements
24: **end for**
25: **return** $\mathcal{N}$

With regard to component interface encoding (Fig. 9, Algorithm 2:2-12), every event emission or reception in a component is translated into a Petri net transition holding the same name as the event but the reversed direction. This transition is connected to specific places that are used to store, using tokens, messages corresponding to the events. For each event emission $c\!:\!a!$ in a component c interface (Fig. 9a), there is a transition for reception in the Petri net ($c\!:\!a?$) and this transition has an output arc to the place where the corresponding message is stored ($??c\!:\!a$). Conversely, for each event reception $c\!:\!a?$ in a component c interface (Fig. 9b), there is a transition for emission in the Petri net ($c\!:\!a!$) and this transition has an input arc from the place where the corresponding message has been stored ($!!c\!:\!a$). The control flow between events in component interfaces is expressed in the Petri net by control places and related arcs connecting the different Petri net transitions. Moreover, tokens are placed in the control places encoding the initial states of the LTS interfaces (Algorithm 2:4) and their evolution will simulate the execution of the entire system.

As far as the contract encoding is concerned (Algorithm 2: 13-24), every synchronous vector is encoded using a tau

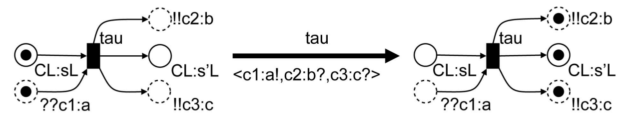

Fig. 10. Encoding pattern for adaptation contracts (and related marking evolution semantics).

[19] transition (Fig. 10, Algorithm 2:16-20) as it represents an internal action of the adaptor. Arcs are added (Algorithm 2:16) to connect these tau transitions in order to enforce their application ordering in the vector LTS. Message transfer is enabled using input/output arcs that connect a tau transition to the places related to the component events involved in the corresponding vector (Algorithm 2:17-20).

We will illustrate further, in this section (Example 6), this encoding into Petri nets on the eMuseum application.

Algorithm 3 generates an adaptor protocol from a set of component LTSs C_i and an adaptation contract. This algorithm, respectively, 1) builds a Petri net encoding for both component LTSs and the contract (Algorithm 3:1), 2) generates the marking graph for this Petri net which contains all of the possible evolutions of the adaptor *wrt* the component LTSs it is in charge of (Algorithm 3:2), and 3) removes remaining deadlocks (*remove_deadlocks*) which correspond to spurious interactions and tau transitions (*reduction*) introduced during the Petri net generation (Algorithm 3:3).

Algorithm 3 *build_adaptor_reordo*
constructs an adaptor with reordering for a set of components given an adaptation contract
inputs components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$ and an adaptation contract (V, L)
output adaptor $Ad = \langle A, S, I, F, T \rangle$
1: $\mathcal{N} := build_PetriNet(\{C_1, \ldots, C_n\}, (V, L))$ // *see Alg. 2*
2: $M := get_marking_graph(\mathcal{N})$
3: $Ad := reduction(remove_deadlocks(M))$
4: **return** Ad

The *reduction* function is used to simplify the adaptor protocols. At this level, several behavioral reductions modulo an equivalence relation can be applied (e.g., $\text{tau}^*.\text{a}$, observational, branching). In our experiments, we used, in particular, a combination of branching and weak trace reductions that enable 1) elimination of tau transitions introduced for message transfer in the encoding of vectors into Petri nets (which are meaningless at the level of the adaptor) while preserving the deadlock freedom property, 2) cutting of similar paths (traces), and 3) determination of the adaptor protocols using a classical automata theory algorithm.

The theoretical complexity of this algorithm mainly lies in the marking graph construction, which is exponential [28]. In practice, it is less expensive as parts of the net are 1-bounded (there is only one token in only one of the places corresponding to the component interface states). We emphasize that the adaptation techniques presented in this section also work for adaptation without reordering. However, since the computational complexity of these

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

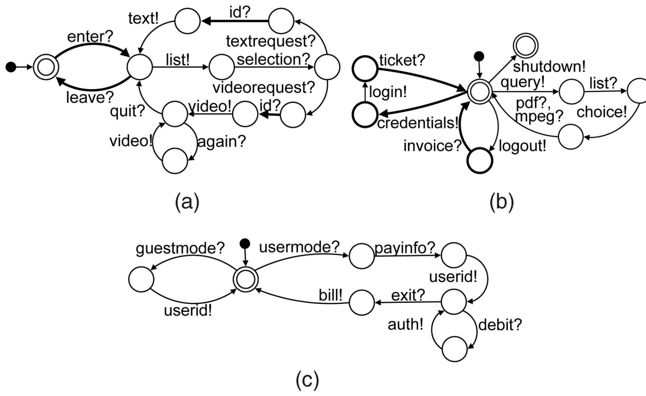

Fig. 11. eMuseum, version 3. (a) ROOM LTS. (b) PDA LTS. (c) SUB LTS.

techniques is greater than those presented in the previous section, they are privileged only if reordering is needed.

Example 6. Let us now describe the last version of the eMuseum application. A third component, a generic pay-per-view subscription server, SUB, is used to manage subscription modes (guest mode for free access and user mode for paying access) and related access identifiers. Upon reception of a registration message (guestmode or usermode), it returns an access identifier (userid message). In case of user registration, reception of the payment information (payinfo message) is required before sending the identifier. Moreover, using debit, the user shopping cart can be updated (with an access authorization sent back each time) before a bill is finally sent (the user account being debited at the same time). There are also changes in new versions of the other two components which are reused. ROOM needs an identifier (id) to be given before information sending in order to update a log file. The access to ROOM is controlled by a signal detecting the entry (enter) and the leaving (leave) of the room. PDA sends payment information (credentials) before logging in and waiting for an acknowledgment (ticket). Finally, after logging out, PDA waits for an invoice of the services it acceded to.

The new corresponding LTSs are given in Fig. 11 (changes are in bold). One may note that

- PDA does not deal with identifiers when doing requests, while ROOM needs them (id?);
- ROOM knows nothing about guest and user modes; and
- reordering is required, first because PDA and ROOM do not support requests in the same way: PDA sends a query before waiting a list of items and selecting one, while ROOM presents its list of items and waits for one to be selected before waiting for either a text or a video request. Moreover, PDA and SUB do not treat the logging in protocol in the same way, the order of the payment information and login request being different in both components.

To work this out, vectors are first defined (differences with the previous example are in bold font). There are two new vectors for the entering and leaving of the system (triggered by the adaptor), while the one for PDA shutdown is reused. In the first case, entering also triggers the guest mode (initial mode):

$$\mathbf{v_{enter}} = \langle \mathbf{ROOM : enter?}, \mathbf{PDA} : _, \mathbf{SUB} : \mathbf{guestmode}? \rangle,$$
$$\mathbf{v_{leave}} = \langle \mathbf{ROOM : leave?}, \mathbf{PDA} : _, \mathbf{SUB} : _ \rangle, \text{ and}$$
$$\mathrm{v_{end}} = \langle \mathrm{ROOM} : _, \mathrm{PDA} : \mathrm{shutdown!}, \mathrm{SUB} : _ \rangle.$$

Vectors for lists and choices are also reused:

$$\mathrm{v_{list}} = \langle \mathrm{ROOM} : \mathrm{list!}, \mathrm{PDA} : \mathrm{list?}, \mathrm{SUB} : _ \rangle \text{ and}$$
$$\mathrm{v_{choice}} = \langle \mathrm{ROOM} : \mathrm{selection?}, \mathrm{PDA} : \mathrm{choice!}, \mathrm{SUB} : _ \rangle.$$

Vectors for entering text (respectively, video) mode and for text (respectively, video) exchange are reused but for two differences: 1) query in PDA now corresponds to requests in ROOM and 2) SUB should be informed about each video being sent:

$$\mathrm{v_{tmode}} = \langle \mathrm{ROOM} : \mathrm{textrequest?}, \mathbf{PDA} : \mathbf{query}!, \mathrm{SUB} : _ \rangle,$$
$$\mathrm{v_{tget}} = \langle \mathrm{ROOM} : \mathrm{text!}, \mathrm{PDA} : \mathrm{pdf?}, \mathrm{SUB} : _ \rangle,$$
$$\mathrm{v_{vmode}} = \langle \mathrm{ROOM} : \mathrm{videorequest?}, \mathbf{PDA} : \mathbf{query}!, \mathbf{SUB : debit?} \rangle,$$
$$\mathrm{v_{vget}} = \langle \mathrm{ROOM} : \mathrm{video!}, \mathrm{PDA} : \mathrm{mpeg?}, \mathbf{SUB} : \mathbf{auth}! \rangle, \text{ and}$$
$$\mathrm{v_{quit}} = \langle \mathrm{ROOM} : \mathrm{quit?}, \mathrm{PDA} : _, \mathrm{SUB} : _ \rangle.$$

Vectors for changing mode are reused and modified to support SUB:

$$\mathrm{v_{user}} = \langle \mathrm{ROOM} : _, \mathrm{PDA} : \mathrm{login!}, \mathbf{SUB} : \mathbf{usermode}? \rangle \text{ and}$$
$$\mathrm{v_{guest}} = \langle \mathrm{ROOM} : _, \mathrm{PDA} : \mathrm{logout!}, \mathbf{SUB} : \mathbf{guestmode}? \rangle.$$

Vectors that support the additional payment relations between PDA and SUB are added:

$$\mathbf{v_{info}} = \langle \mathbf{ROOM} : _, \mathbf{PDA} : \mathbf{credentials}!, \mathbf{SUB} : \mathbf{payinfo}? \rangle,$$
$$\mathbf{v_{bill}} = \langle \mathbf{ROOM} : _, \mathbf{PDA} : \mathbf{invoice}?, \mathbf{SUB} : \mathbf{bill}! \rangle, \text{ and}$$
$$\mathbf{v_{exit}} = \langle \mathbf{ROOM} : _, \mathbf{PDA} : _, \mathbf{SUB} : \mathbf{exit}? \rangle.$$

Identifier exchange is finally specified with three vectors (one for guest mode, one for user mode, and one for resending):

$$\mathbf{v_{gid}} = \langle \mathbf{ROOM} : \mathbf{id}?, \mathbf{PDA} : _, \mathbf{SUB} : \mathbf{userid}! \rangle,$$
$$\mathbf{v_{uid}} = \langle \mathbf{ROOM} : \mathbf{id}?, \mathbf{PDA} : \mathbf{ticket}?, \mathbf{SUB} : \mathbf{userid}! \rangle, \text{ and}$$
$$\mathbf{v_{reid}} = \langle \mathbf{ROOM} : \mathbf{id}?, \mathbf{PDA} : _, \mathbf{SUB} : _ \rangle.$$

Vector $\mathrm{v_{again}}$ is left over, suppressing the possibility of video resending.

As for the previous example, we may now use a vector LTS to specify their possible orderings. We propose two different contracts: one supporting only the GUEST mode (Fig. 12a) and one supporting both modes (Fig. 12b).

The contract for the GUEST mode (Fig. 12a) focuses on what happens between when one enters and when one leaves the room. Moreover, it specifies that, once the identifier has been first exchanged, the identifier is resent by the adapter (vector $\mathbf{v_{reid}}$) only if a new query happens (vector $\mathbf{v_{tmode}}$). But for these two constraints, the contract is not restrictive and does not specify any

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

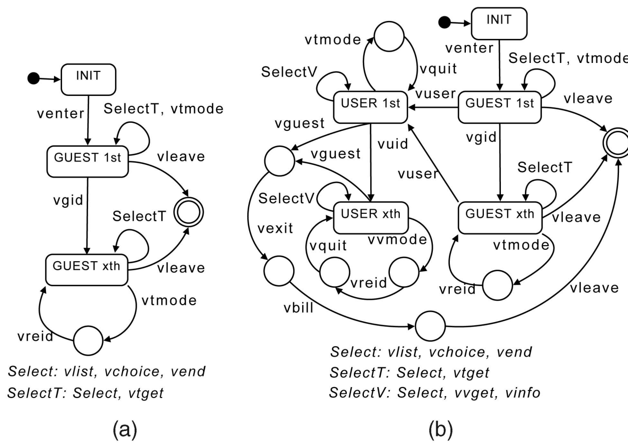

Fig. 12. Adaptation contracts for eMuseum, version 3. (a) GUEST mode. (b) Full mode.

particular ordering of vectors. The adaptation process will, therefore, find all possible ones such that the adapted system does not deadlock. The contract for the full mode (Fig. 12b) adds a part relative to the USER mode. One may note that it is symmetric to the GUEST mode contract but with some differences. We must first take into account the quit message emission by the adaptor (vector v_{quit}) to avoid blocking once a video has been exchanged. This is put into practice, e.g., by adding this vector in the vector LTS at the end of the v_{tmode} loops (twice). Moreover, while passing from GUEST to USER mode is quite simple (vector v_{user}), leaving USER mode should also take into account the final payment using vectors v_{bill} and v_{exit}. This is representative of one-to-many correspondence, here between logout in PDA and both exit and guestmode in SUB. The obtaining of the full mode contract (and the difference between the USER and the GUEST modes) has been achieved in several steps, using postgeneration adaptor assessment (see support for contract design in Section 8). In the sequel, we will present our approach on the first contract due to the complexity of the adaptor for the full mode.

The Petri net generated for this example is given in Fig. 13. To help the reader, we present the parts of the Petri net which are generated for ROOM, PDA, SUB, and the contract separately. The nets are glued on dashed places, accept transitions, and, for the contract, on vector transitions.

The adaptor for the GUEST mode has 204 states and 404 transitions (494 states and 1,101 transitions before pruning paths to deadlocks). After reduction, the resulting final adaptor has 52 states and 104 transitions (Fig. 14, where the initial state is in light gray and the final states are in black). We emphasize that it is much simpler to give an adaptation contract and use our automatic adaptor protocol generation techniques than to directly write the protocol by hand.

One may note different things (see Fig. 14 zoom):

- name mismatch is solved, e.g., choice in PDA versus selection in ROOM;
- messages are reordered when required, e.g., PDA sending query and then waiting for the list of possible information to be displayed, while ROOM sending first the list and waiting for a selection before waiting for either a textrequest or videorequest which correspond to query;
- id is resent to ROOM when required;
- the contract is permissive—e.g., *wrt* in which order to apply vectors v_{list}, v_{choice}, v_{tmode}, v_{tget}, and v_{vend}—and the adaptor contains all possible orderings not leading to deadlocks (yet we have only represented one possible ordering on the zoom).

The adaptor for the full mode has 1,477 states and 3,326 transitions (2,719 states and 6,464 transitions before pruning paths to deadlocks). After reduction, the resulting final adaptor has 307 states and 627 transitions. Due to its size, it is not given here. Performing verification on the adapted system (made up of the components and the adaptor), we have been able to check with CADP that important system-level properties are enforced through adaptation: 1) No video is sent before the PDA logs on and 2) a debit is performed for each video being sent.

6 The Adaptor Tool

The approach for software adaptation that we presented in the previous sections of this paper was implemented in a tool called Adaptor [29]. The kernel of Adaptor corresponds to the implementation of the algorithms that generate adaptor protocols being given behavioral interfaces of components and an adaptation contract. In addition, Adaptor presents graphical interfaces to load and visualize the different inputs, to apply the different adaptation steps, and to visualize the intermediate encodings and final results. The tool was initially developed in Python (about 9,000 lines of code)[1] and uses GTK+ technology for the development of the user interface. More recently, to simplify the access and use of the tool, a Web service version of Adaptor (WS-Adaptor) was implemented in Java. It enables one to adapt component protocols without installing more than a GUI client (the engine and the required dependencies running in the distant Web service host).

Different input and output formats are used to describe, respectively, interfaces of components, contracts, and resulting adaptors. With regard to inputs, LTS interfaces may be described using XML or the Aldebaran textual format [22] (file extension .aut). Vectors and vector LTSs involved in contracts are specified using XML.

Once the inputs are loaded, Adaptor uses dot [30] (graphviz) to visualize interfaces of components, intermediate results for contracts, Petri nets, and adaptors. Textual formats are also possible for visualization or storing and analysis purposes, namely, .aut for LTSs and .net for Petri nets. Adaptor interacts with two other external tools, namely, TINA and CADP. TINA [31] is a tool to design and validate Petri nets. It allows the application of structural

1. Approximately 5,000 lines of code correspond to the encoding of the adaptation techniques, and approximately 4,000 lines to the user interface.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

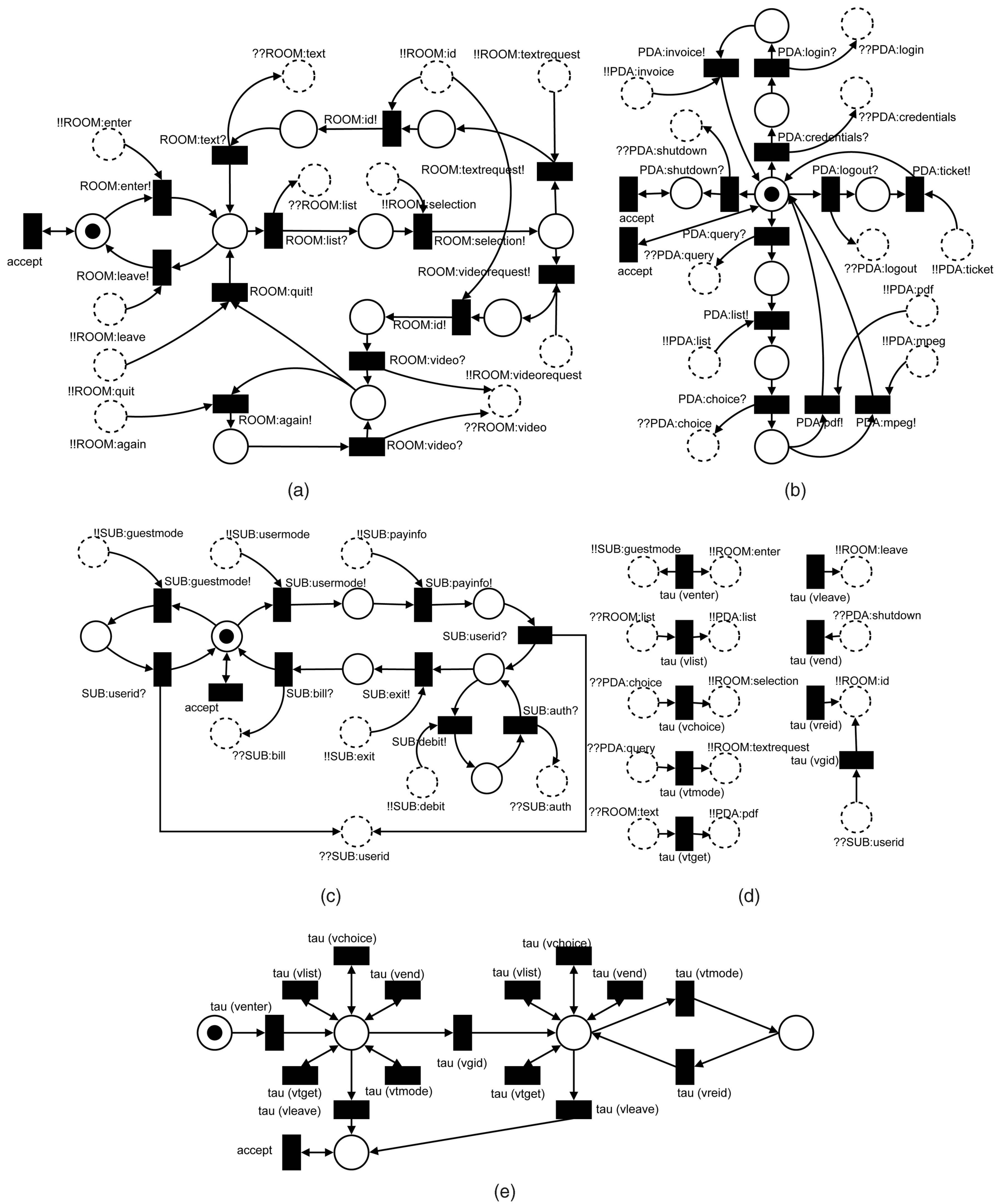

Fig. 13. Petri net encoding for eMuseum, version 3 (GUEST mode). (a) ROOM encoding. (b) PDA encoding. (c) SUB encoding. (d) Contract encoding (vectors). (e) Contract encoding (vector LTS).

and reachability analysis on Petri nets. TINA is used in Adaptor to compute marking graphs from Petri nets encodings. CADP [22] is a toolbox to verify concurrent systems. It is used to compute the mismatch test using its EXP.OPEN tool and to perform reductions of the adaptor LTSs using BCG_MIN and Reductor.

The current version of Adaptor fully supports transactional components. For nontransactional ones, avoiding state explosion when computing marking graphs requires that messages cannot be infinitely generated. This means, first, that a component should not send some message infinitely and independently (i.e., without having this action triggered by a message reception or requiring an acknowledgment). In the same way, the adaptor should not infinitely and independently generate messages using vectors such as $\langle _, \ldots, c:m?, _, \ldots \rangle$.

Adaptor has been used to generate the adaptor protocols presented in this paper, but it has been validated and applied to many other examples as well (approximately 70 examples which correspond to 25,000 lines of XML specification) such as a Video-On-Demand service, a pervasive music player, a library lending system, and

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

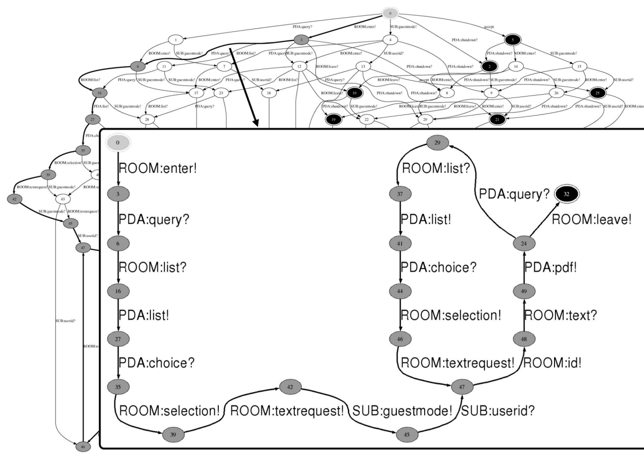

Fig. 14. Adaptor protocol for eMuseum, version 3 (GUEST mode).

several simpler client-server systems. More details are available on the Adaptor Web page [29].

We show in Fig. 15 three screenshots of Adaptor to give a flavor of what the tool looks like, here applied to eMuseum. The Adaptor GUI is made up of three different windows: The left-hand side window contains the already-loaded component interfaces and contracts, the right-hand side window is used to visualize all the elements involved in the adaptation process (interfaces, contracts, Petri nets, adaptors) under different formats (graphical, textual, XML), and the bottom window is the console window. The first screenshot (Fig. 15a) shows the SUB component LTS. The second one (Fig. 15b) is a textual description of the contract we have presented in Example 6. Finally, the last one (Fig. 15c) shows a piece of the adaptor during the adaptation process.

7 Related Work

Software composition and adaptation is currently a hot topic in software engineering research. A quick look over the Web will easily produce a great number of works—ranging from deep theoretical works (e.g., [32], which uses category theory for signature adaptation via *superposition*, or name morphisms) to more practical proposals (e.g., [33] for Web Services). Furthermore, an increasing number of events are specifically focused on adaptation or have it as one of their main topics (e.g., the WCAT series of workshops [34], starting in 2004).

The issues related to software component integration have been a classical field of study in software engineering and component mismatch has been described at all the levels of interoperability. A taxonomy of interface mismatches appears in [2], classifying them into *technical* mismatch, coming from the use of different operating systems, platforms, and frameworks; *signature* mismatch, related to different names of methods and services, parameter and exception types, and parameter ordering; *protocol* or *behavioral* mismatch, caused by different message ordering and the absence or surplus of messages; *quality of service* mismatch, linked to different assumptions on properties like security, persistency, reliability, or efficiency; and, finally, *conceptual* or *semantic* mismatch, coming from the use of homonyms, synonyms for describing the services provided, or the existence of sub and superordination relations between services.

Although some practical issues related to technical interoperability between different platforms still remain, we consider that these do not demand a significant research effort. Accordingly, the research in the field has recently begun to explore the rest of the sources of mismatch mentioned above. In particular, in this work, the focus on both the signature and behavioral levels, where the use of formal notations based on logic formulas, Petri nets, process algebras, state machines, and many others has been promoted for enhancing software interfaces with a behavioral description (see [35] for an early instance). One of the first proposals for defining behavioral mismatch from a formal point of view can be found in [8], where process algebra is used for specifying and reasoning about software composition. The work is continued in [36], where a means of characterizing connector wrappers as protocol transformations and reasoning about their properties is presented. A similar approach is presented in [13], where

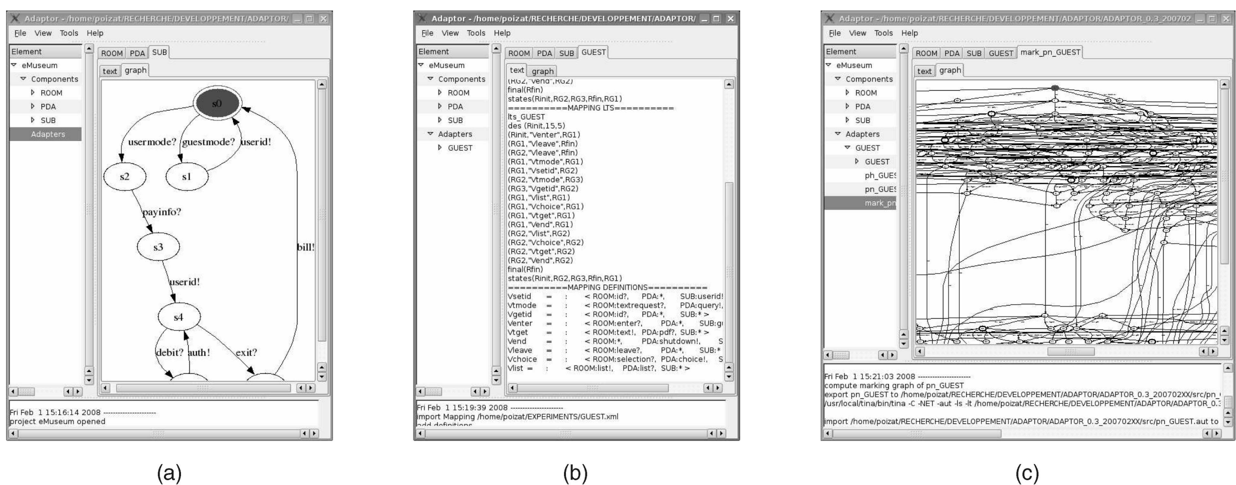

Fig. 15. Screenshots of the Adaptor tool—eMuseum, version 3. (a) SUB component LTS. (b) Contract (textual). (c) Adaptor before reduction (part of).

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

compatibility and substitutability are defined in the context of CORBA as a first attempt to put these ideas into industrial practice.

However, once behavioral mismatch is detected, the issue of how to adapt component protocols in order to solve it arises. Many of the approaches found in the literature work at the implementation level, some of them [37], [38], [39] related to existing programming languages and platforms, such as BPEL or SCA components, and suggesting manual or at most semiautomated techniques for solving behavioral mismatch. For instance, [37] describes a model-based approach for verifying Web service compositions, including the verification of properties created from design specifications and implementation models to confirm expected results. However, once a violation of the properties is detected, it should be manually corrected, either in the implementation of the components or in the specification models, as part of an iterative development process. Also in the context of Web services and BPEL, [38] outlines a methodology for the generation of adaptors capable of solving behavioral mismatches between BPEL processes. In their adaptation methodology, the authors use an intermediate workflow language for describing component behavioral interfaces and they use lock analysis techniques to detect behavioral mismatch. Similarly, [39] provides automated support for the identification of protocol-level mismatches, but is able to generate an adaptor only in the absence of deadlock. If deadlock may arise from the combination of the components, the authors propose a way to handle the situation by generating a tree for all mismatches that result in a deadlock and suggesting some hints for assisting the designer in the manual implementation of the actual adaptor.

Current approaches aiming to provide a fully automated solution to this problem are comparatively fewer and can be divided into *restrictive*, *generative*, and ad hoc [4]. Restrictive approaches [26], [40], [41], [42], [43] simply try to solve the problem by cutting off the behavior that may lead to mismatch, thus restricting the functionality of the components involved. On the contrary, generative approaches like [9], [44], [45] try to accommodate the protocols without restricting the behavior of the components, by generating adaptors that act as mediators, remembering and reordering events and data when necessary. Finally, ad hoc approaches (see, for instance, [46], [47], [48]) do not address the adaptation from a general, automatable point of view, but propose specific practical solutions for particular situations instead.

The foundation for automatic behavioral adaptation was set by Yellin and Strom (YS). In their seminal article [9], they introduced formally the notion of *adaptor* as a software entity capable of enabling the interoperation of two components with mismatching behavior. They used finite state machines to specify component interaction, to define a relation of compatibility, and to address the task of (semi)automatic adaptor construction following the generative approach mentioned above.

More recently, Schmidt and Reussner (SR) presented a particular adaptation approach as a solution to synchronization problems between concurrent components [46]. The proposal addresses, for instance, situations where one component is accessed simultaneously by two other components. The approach is based on algorithms close to the synchronous products we use in this paper. Moreover, they can solve protocol incompatibilities, enabling one of the involved components to perform several communication actions before or after synchronizing with its partners. These ideas are implemented in the CoConut/J tool suite [49], where the authors introduce the concept of parameterized contracts and a model for component interfaces. This paper also presents algorithms and tools for specifying and analyzing component interfaces in order to check interoperability and to generate adapted component interfaces automatically. In comparison, our proposal is more general and based on a rich notation to deal with possibly complex adaptation scenarios, whereas the SR approach works out only precise situations in which mismatch may happen, without using any contract language for adaptor specification.

In their paper *Adapt or Perish* [50], Dumas et al. presented an approach to behavioral interface adaptation based on the definition of a set of adaptation operations for establishing the basic relation patterns between the message names used in the components being adapted and they defined a trace-based algebra for describing the transformations required to solve the adaptation problem. They also present a visual notation for describing a mapping between the behavioral interfaces of the components. Their approach is similar to ours in the sense that these basic operations correspond to the different relations (one-to-one, one-to-many, many-to-one, one-to-zero, etc.) between message names that can be defined by means of our synchronous vectors. However, their proposal does not present a solution for deriving an adaptor from the visual mappings, but just contains a preliminary (i.e., nonsufficient) condition for detecting deadlock scenarios in the behavioral interfaces. Moreover, their mappings require relating the messages at the behavioral level (i.e., matching messages directly from the component protocol specifications), while our adaptation contracts are more abstract since the mapping is performed at the signature level (i.e., between the messages declared in the component interfaces) from which we automatically obtain an adaptor solving the mismatch at the behavioral level. Finally, their approach is not able to perform message reordering when it is required for solving the problem.

Taking the YS proposal as a starting point, the work of Brogi et al. (BBCP) [44], [45] presents a methodology for generative behavioral adaptation. In their proposal, component behavior is specified using a process algebra—a subset of the π-calculus—where service offering/invocation is represented by input/output events in the calculus, respectively. The starting point of their adaptation process is a *mapping*, an adaptation contract that states correspondences between the services of the components being adapted. Then, an adaptor generation algorithm refines the specification given by the mapping into a concrete adaptor implementation, taking into account the behavioral interfaces of the components, which ensures correct interaction between them according to the mapping. The adaptor is able to accommodate not only signature mismatch between

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

service names but also behavioral mismatch (i.e., the interaction protocols that the components follow or the partial ordering in which services are offered/invoked).

Another interesting proposal in this field is that of Inverardi and Tivoli (IT) [26]. Certain aspects of their work go beyond BBCP by addressing how to enforce certain behavioral properties (namely, liveliness and safety properties expressed as specific processes) out of a set of already implemented behaviors. Starting from the specification with MSCs of the components to be assembled and of the properties that the resulting system should verify, they automatically derive the adaptor glue code for the set of components in order to obtain a property-satisfying system. In order to do that, they follow the so-called restrictive approach. The IT proposal was extended in [40] with the use of temporal logic; coordination policies are expressed as LTL properties and then translated into Büchi automata. Recent outcomes of this research line allow a distributed implementation of the adaptors [41] and take into account time and other QoS issues [42].

Another example of the restrictive approach is the work of de Alfaro et al. [11], [43], who use game theory to achieve behavioral adaptation. One of the relevant features of the proposal is that time information can be taken into account within the component interfaces.

Our approach to behavioral adaptation can be considered as *both generative and restrictive* since we address behavioral adaptation by enabling event reordering (as in BBCP), while we also remove incorrect behavior (as in IT). Similarly to both of them, our main goal is to ensure deadlock freedom. However, more complex adaptation policies and properties can be specified by means of our vector LTSs. A deeper comparison with the aforementioned approaches yields that our proposal addresses system-wide adaptation (i.e., differently from BBCP, it may involve more than two components) and that it is based on LTS descriptions of component behavior instead of using process algebra as in BBCP. However, we may also describe behavior by means of a simple process algebra and use its operational semantics to derive LTSs from it [14]. Differently from IT, which requires name matching, we use synchronous vectors in our adaptation contracts, playing a similar function as the mapping rules in BBCP. With that, we are able to perform adaptation of incompatible events. Finally, our approach is fully tool equipped, while BBCP have only presented a sketch of the implementation of their adaptation algorithm.

Nevertheless, the most relevant achievement of our current proposal is the use of vector LTSs for imposing additional properties over adaptation contracts. In fact, the semantics of BBCP mappings can be expressed by combining their different rules in a vector LTS with a single state and all vector transitions looping on it. On the contrary, our vector LTSs are much more expressive, solving the problem of BBCP underspecified mappings [44] and allowing us to take into account a new class of adaptation problems.

A different characterization of behavioral adaptation techniques may classify them into *immutable* and *contextual*. Immutable approaches are those that define a static set of rules for describing the adaptation required and these rules are applied uniformly during the whole adaptation process. On the contrary, contextual adaptation pays attention to context information in order to decide on-the-fly the adaptation strategy to apply. Our present approach allows contextual adaptation by the use of vector LTSs which govern when the adaptation rules are applied (as shown in Figs. 6 and 12), while the rest of the approaches mentioned above are static. Some recent works based on the BBCP proposal try to address more flexible ways of contextual adaptation [51].

Finally, most of the current adaptation proposals—and our present work among them—may be considered as *global* since they proceed by computing global adaptors for closed systems made up of a predefined and fixed set of components. However, this is not satisfactory when the system may evolve, with components entering or leaving it at any time, e.g., for pervasive computing. To enable adaptation on such systems, an *incremental* approach should be considered by which the adaptation is dynamically reconfigured depending on the components present in the system. One of the first attempts in this direction is [52], whose proposal for incremental software construction by means of refinement allows for simple signature adaptation. However, to our knowledge, the only proposal addressing incremental adaptation at the behavioral level is [53].

8 Concluding Remarks

Software Adaptation is widely accepted as a promising solution to favor the reuse of black-box components that require nonintrusive adjustments to make them fit with the specificities of the system-to-be. In this paper, we have presented a proposal for software adaptation at the signature and behavioral levels based on a simple adaptation contract notation. These contracts can be used to express correspondences (possibly involving mismatching messages) not only between an arbitrary number of components but also complex adaptation scenarios. Our proposal is equipped with two algorithms, depending on whether reordering is necessary or not in the adaptation process. The first one is based on synchronous product computation and the second one on encodings into Petri nets. Our proposal is completely supported by a tool which was applied to many examples.

In this paper, we follow a regular model-based approach, focusing on abstract (platform-independent) behavioral interface models, LTSs. It has been demonstrated, usually for verification purposes, that such abstract models could be derived from existing implementation platforms' languages, e.g., [54], [55], [56] for Web services. With regard to adaptation, model-based behavioral adaptation has been applied to COM/DCOM components in [26] and to Web services in [38], [39]. In a recent paper [16], we have addressed WF components. We have shown how LTS descriptions could be extracted automatically from WF workflows and how a new WF component could be obtained from an adaptor protocol generated with the techniques we have presented here. Therefore, we think the proposed adaptation techniques are of great interest for real-world software components or Web services.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

There are still some open issues in our proposal deserving future work. In this part of the conclusion, we will particularly emphasize three perspectives, namely, data adaptation, contract design support, and application to pervasive systems.

Data adaptation. Taking data exchange into account in protocols is important to ensure full compatibility. So far, this can be supported in the approach at hand using additional messages for data exchange in the abstract component protocols (LTSs), as presented in Section 2.1. Provided this encoding is performed as a preprocessing and the adaptation contract takes the additional messages into account, the protocols can be adapted, as demonstrated in [16].

Directly supporting data types would be more efficient but would first require more expressive models than LTSs. In particular, we consider Symbolic Transition Systems (STSs) [57] or Extended State Diagrams [58] as good candidates since they allow the description of the data involved in the operations within the protocol without suffering from the state explosion problem. Then, data types should also be taken into account in the contract specification as well as for the additional message encoding technique, above. As far as the adaptation process itself is concerned, we are studying two possible techniques. The first one is compatible with the approach at hand, e.g., for the reordering approach, it consists of taking the data types into account in the Petri net encoding patterns (data type resources being generated for component emissions, data type resources being consumed for components receptions, and data types being transferred for data vectors). We are currently looking for an efficient Petri net encoding, using Petri net extensions, in order to avoid state explosion problems. The second technique specifically addresses these efficiency issues. It consists of implementing data adaptation separately from the message-based one through a data adaptation engine that would be embedded in the adaptor implementation and that would store received values and redistribute them *wrt* the correspondences expressed in the data contract.

Support for contract design. The design of an adaptation contract may be a nontrivial and error-prone task, leading to too many interactions being removed in the adaptation process to ensure deadlock-freedom. To address this issue, recent work has focused on postgeneration adaptor assessment, either by reusing existing model checkers [59] or by developing new tools such as Clint [60], which is able to graphically represent deadlocks in components and interactions that are removed in the adaptation process. The former approach is more powerful, yet it requires temporal logic formulas. This is the approach we have used in this paper to obtain our mappings. The latter is less expressive (as far as the kind of properties which are assessed over the adapted system are concerned), yet it benefits from being fully automatic.

Approaches dedicated to the automatic generation of compositions are indeed the current goal of research groups working at the semantic interoperability level, e.g., adding semantic annotation to (Web) services [61]. Yet, enforcing a semantic description for all components (including legacy ones) is a strong assumption.

We are convinced that an assisted design approach is a good trade-off between complete automation and manual writing of the composition and adaptation contracts. Further, it enables a user composition vision [62], [63]. As a perspective, we plan to propose techniques to support the contract design task. A partial specification of the contract could be given for which the remaining composition issues (such as deadlocks in components and interactions that would be removed by the adaptation process) would be emphasized using Clint. In addition, incremental contract construction, where, at each step, possible message correspondences to complete the contract would be proposed, would foster the user-friendliness of the contract design process.

Self-adaptive pervasive systems. A perspective in the context of funded research projects is to apply our adaptation techniques to pervasive systems. In this field, self-adaptation is a mandatory feature because fewer assumptions can be done on the system at hand, e.g., new components or services can show up or disappear at runtime, while the overall adaptation mechanism should support these evolutions and keep on making the system work in a reliable way. We are currently exploring Dynamic Aspect Oriented Programming to put into practice adaptation techniques in this highly dynamic context.

ACKNOWLEDGMENTS

The authors thank Sandrine Beauche and Juan David G. Urbano for their participation in the implementation of Adaptor, as well as Bernard Berthomieu, Frédéric Lang, Massimo Tivoli, and François Vernadat for their help on external tool support, interesting comments, and fruitful discussions. The authors are also grateful to the anonymous referees, whose comments helped a lot to improve this paper. This work is partially supported by the projects "PERvasive Service cOmposition" (PERSO), funded by the French National Agency for Research (ANR-07-JCJC-0155-01), TIN2004-07943-C04-01, funded by the Spanish Ministry of Education and Science (MEC), and P06-TIC-02250, funded by the Andalusian Local Government.

REFERENCES

[1] O. Nierstrasz and T.D. Meijler, "Research Directions in Software Composition," *ACM Computing Surveys*, vol. 27, no. 2, pp. 262-264, 1995.

[2] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, "Towards an Engineering Approach to Component Adaptation," *Architecting Systems with Trustworthy Components.* pp. 193-215, Springer, 2006.

[3] C. Canal, J.M. Murillo, and P. Poizat, "Coordination and Adaptation Techniques for Software Entities," *Proc. European Conf. Object-Oriented Programming '04 Workshop Reader,* pp. 133-147, 2004.

[4] C. Canal, J.M. Murillo, and P. Poizat, "Software Adaptation," *L'Objet,* special issue on coordination and adaptation techniques, vol. 12, no. 1, pp. 9-31, 2006.

[5] G. Agha, "Introduction to Special Issue on Adaptive Middleware," *Comm. ACM*, vol. 45, no. 6, pp. 30-64, June 2002.

[6] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

[7] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau, "Making Components Contract Aware," *Computer*, vol. 32, no. 7, pp. 38-45, July 1999.

[8] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 3, pp. 213-249, 1997.

[9] D.M. Yellin and R.E. Strom, "Protocol Specifications and Components Adaptors," *ACM Trans. Programming Languages and Systems*, vol. 19, no. 2, pp. 292-333, 1997.

[10] J. Magee, J. Kramer, and D. Giannakopoulou, "Behaviour Analysis of Software Architectures," *Proc. First Working IFIP Conf. Software Architecture*, pp. 35-49, 1999.

[11] L. de Alfaro and T.A. Henzinger, "Interface Automata," *Proc. Joint Eighth European Software Eng. Conf. and Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 109-120, 2001.

[12] F. Plasil and S. Visnovsky, "Behavior Protocols for Software Components," *IEEE Trans. Software Eng.*, vol. 28, no. 11, pp. 1056-1076, Nov. 2002.

[13] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo, "Adding Roles to CORBA Objects," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 242-260, Mar. 2003.

[14] C. Canal, P. Poizat, and G. Salaün, "Synchronizing Behavioural Mismatch in Software Composition," *Proc. Eighth IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems*, pp. 63-77, 2006.

[15] C. Canal, P. Poizat, and G. Salaün, "Model-Based Adaptation of Behavioural Mismatching Components (Long Version)," http://www.ibisc.univ-evry.fr/~poizat, 2008.

[16] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat, "A Model-Based Approach to the Verification and Adaptation of WF/.NET Components," *Proc. Fourth Int'l Workshop Formal Aspects of Component Software (FACS '07)*, pp. 39-55, 2008.

[17] K. Scribner, *Microsoft Windows Workflow Foundation: Step by Step.* Microsoft Press, 2007.

[18] *Handbook of Process Algebra*, J.A. Bergstra, A. Ponse, and S.A. Smolka, eds. Elsevier, 2001.

[19] R. Milner, *Communication and Concurrency.* Prentice Hall, 1994.

[20] *Formal Methods for Software Architectures*, M. Bernardo and P. Inverardi, eds. Springer, 2003.

[21] A. Arnold, *Finite Transition Systems.* Prentice Hall, 1994.

[22] H. Garavel, R. Mateescu, F. Lang, and W. Serwe, "CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes," *Proc. 19th Int'l Conf. Computer Aided Verification*, pp. 158-163, 2007.

[23] S. Haddad and P. Poizat, "Transactional Reduction of Component Compositions," *Proc. 27th IFIP Int'l Conf. Formal Methods for Networked and Distributed Systems*, pp. 341-357, 2007.

[24] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[25] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioural Models from Scenarios," *IEEE Trans. Software Eng.*, vol. 29, no. 2, pp. 99-115, Feb. 2003.

[26] P. Inverardi and M. Tivoli, "Deadlock Free Software Architectures for COM/DCOM Applications," *J. Systems and Software*, vol. 65, no. 3, pp. 173-183, 2003.

[27] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541-580, 1989.

[28] C. Rackoff, "The Covering and Boundedness Problems for Vector Addition Systems," *Theoretical Computer Science*, vol. 6, pp. 223-231, 1978.

[29] "Adaptor, December 2007 Distribution (LGPL Licence)," http://www.ibisc.univ-evry.fr/~poizat, 2008.

[30] E. Gansner, E. Koutsofios, and S. North, *DOT User's Manual*, Jan. 2006.

[31] B. Berthomieu, P.-O. Ribet, and F. Vernadat, "The Tool TINA—Construction of Abstract State Spaces for Petri Nets and Time Petri Nets," *Int'l J. Production Research*, vol. 42, no. 14, pp. 2741-2756, 2004.

[32] M. Wermelinger, A. Lopes, and J.L. Fiadeiro, "A Graph Based Architectural (Re)Configuration Language," *Proc. Joint Eighth European Software Eng. Conf. and Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.*, pp. 20-32, 2001.

[33] S. Dustdar and W. Schreiner, "A Survey on Web Services Composition," *Int'l J. Web and Grid Services*, vol. 1, no. 1, pp. 1-30, 2005.

[34] *Proc. Int'l Workshop Series on Coordination and Adaptation Techniques*, http://wcat.unex.es, 2008.

[35] D. Lea and J. Marlowe, "Interface-Based Protocol Specification of Open Systems Using PSL," *Proc. Ninth European Conf. Object-Oriented Programming*, pp. 374-398, 1995.

[36] B. Spitznagel and D. Garlan, "A Compositional Formalization of Connector Wrappers," *Proc. 25th Int'l Conf. Software Eng.*, pp. 374-384, 2003.

[37] H. Foster, S. Uchitel, J. Maggee, and J. Kramer, "Model-Based Verification of Web Service Compositions," *Proc. 18th IEEE Int'l Conf. Automated Software Eng.*, pp. 152-163, 2003.

[38] A. Brogi and R. Popescu, "Automated Generation of BPEL Adapters," *Proc. Fourth Int'l Conf. Service Oriented Computing*, pp. 27-39, 2006.

[39] H.R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-Automated Adaptation of Service Interactions," *Proc. 16th Int'l World-Wide Web Conf.*, pp. 993-1002, 2007.

[40] P. Inverardi and M. Tivoli, "Software Architecture for Correct Components Assembly," *Formal Methods for Software Architectures* pp. 92-121, Springer, 2003.

[41] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli, "SYNTHESIS: A Tool for Automatically Assembling Correct and Distributed Component-Based Systems," *Proc. 29th Int'l Conf. Software Eng.*, pp. 784-787, 2007.

[42] M. Tivoli, P. Fradet, A. Girault, and G. Goessler, "Adaptor Synthesis for Real-Rime Components," *Proc. 13th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems*, pp. 185-200, 2007.

[43] L. de Alfaro and M. Stoelinga, "Interfaces: A Game-Theoretic Framework to Reason about Open-Systems," *Proc. Second Int'l Workshop Foundations of Coordination Languages and Software Architectures*, pp. 3-23, 2004.

[44] A. Bracciali, A. Brogi, and C. Canal, "A Formal Approach to Component Adaptation," *J. Systems and Software*, vol. 74, no. 1, pp. 45-54, 2005.

[45] A. Brogi, C. Canal, and E. Pimentel, "Component Adaptation through Flexible Subservicing," *Science of Computer Programming*, vol. 63, no. 1, pp. 39-56, 2006.

[46] H.W. Schmidt and R.H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronization," *Proc. Fifth Int'l Conf. Formal Methods for Open Object-Based Distributed Systems*, pp. 213-229, 2002.

[47] H. Min, S. Choi, and S. Kim, "Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition," *Proc. Seventh Int'l Symp. Component-Based Software Eng.*, pp. 40-47, 2004.

[48] B. Benatallah, F. Casati, D. Grigori, H.R. Motahari-Nezhad, and F. Toumani, "Developing Adapters for Web Services Integration," *Proc. 17th Conf. Advanced Information Systems Eng.*, pp. 415-429, 2005.

[49] R.H. Reussner, "Automatic Component Protocol Adaptation with the CoConut/J Tool Suite," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 627-639, 2003.

[50] M. Dumas, K.W.S. Wang, and M.L. Spork, "Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation," *Proc. Fourth Int'l Conf. Business Process Management*, pp. 65-80, 2006.

[51] J. Cubo, G. Salaün, J. Cámara, C. Canal, and E. Pimentel, "Context-Based Adaptation of Component Behavioural Interfaces," *Proc. Ninth Conf. Coordination Models and Languages*, pp. 305-323, 2007.

[52] R.J. Back, "Incremental Software Construction with Refinement Diagrams," Technical Report 660, Turku Center for Computer Science, 2005.

[53] P. Poizat and G. Salaün, "Adaptation of Open Component-Based Systems," *Proc. Ninth IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems*, pp. 141-156, 2007.

[54] X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services," *Proc. 13th Int'l Conf. World Wide Web*, pp. 621-630, 2004.

[55] H. Foster, S. Uchitel, and J. Kramer, "LTSA-WS: A Tool for Model-Based Verification of Web Service Compositions and Choreography," *Proc. 28th Int'l Conf. Software Eng.*, pp. 771-774, 2006.

[56] G. Salaün, L. Bordeaux, and M. Schaerf, "Describing and Reasoning on Web Services Using Process Algebra," *Int'l J. Business Process Integration and Management*, vol. 1, no. 2, pp. 116-128, 2006.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.

[57] O. Maréchal, P. Poizat, and J.-C. Royer, "Checking Asynchronously Communicating Components Using Symbolic Transition Systems," *Proc. Int'l Symp. Distributed Objects and Applications,* pp. 1502-1519, 2004.
[58] C. Attiogbé, P. Poizat, and G. Salaün, "A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes," *IEEE Trans. Software Eng.,* vol. 33, no. 3, pp. 157-170, Mar. 2007.
[59] P. Poizat, G. Salaün, and M. Tivoli, "An Adaptation-Based Approach to Incrementally Build Component Systems," *Proc. Third Int'l Workshop Formal Aspects of Component Software (FACS '06),* pp. 155-170, 2007.
[60] J. Cámara, G. Salaün, and C. Canal, "Clint: A Composition Language Interpreter," *Proc. 11th Int'l Conf. Fundamental Approaches to Software Eng.*, pp. 423-427, 2008.
[61] S. Ben Mokhtar, N. Georgantas, and V. Issarny, "COCOA: COnversation-Based Service Composition in PervAsive Computing Environments with QoS Support," *J. Systems and Software,* special issue on ICPS '06, vol. 80, no. 12, pp. 1941-1955, 2007.
[62] M. Burnett, C. Cook, and G. Rothermel, "End-User Software Engineering," *Comm. ACM,* vol. 47, no. 9, pp. 53-58, Sept. 2004.
[63] X. Liu, G. Huang, and H. Mei, "Towards End User Service Composition," *Proc. 31st Ann. Int'l Computer Software and Applications Conf.,* pp. 676-678, 2007.

Carlos Canal received the PhD degree in computer science from the University of Málaga, Spain, in 2001, where he is currently an associate professor of software engineering. His current research interests include component-based software development and software adaptation, in particular with the application of formal methods to the specification, safe composition, and adaptation of interacting software components.

Pascal Poizat received the PhD degree in computer science from the University of Nantes, France, in 2000. He is currently an associate professor at the University of Évry, France, and an invited researcher on the ARLES project-team at INRIA, France. His current research interests include formal models and verification techniques for component and service-based systems, more specifically issues related to coordination and adaptation. He is a member of the IEEE Computer Society.

Gwen Salaün received the PhD degree in computer science from the University of Nantes, France, in 2003. He is currently an associate researcher at the University of Málaga, Spain. His current research interests include issues related to formal methods, specification and verification, and composition and adaptation of components and services.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: UR Rhône Alpes. Downloaded on January 15, 2009 at 06:57 from IEEE Xplore. Restrictions apply.