INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Behavioral Adaptation of Component Compositions based on Process Algebra Encodings

Radu Mateescu — Pascal Poizat — Gwen Salaün

N° 6362

Novembre 2007

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR-6362--FR+ENG

Behavioral Adaptation of Component Compositions based on Process Algebra Encodings

Radu Mateescu* , Pascal Poizat** , Gwen Salaün***

Thème COM — Systèmes communicants

Projets Vasy et Arles

Rapport de recherche n° 6362 — Novembre 2007 — 25 pages

Abstract: Component-Based Software Engineering and Service Oriented Computing promote the reuse of existing software entities, respectively components and services. Being developed independently, these entities often mismatch. Software adaptation has been proposed as a solution to this issue with the objective to generate, as automatically as possible, adaptors, *i.e.*, software pieces solving mismatch in a non-intrusive way. We propose an approach for the generation of adaptor protocols from component behavioral interfaces and composition contracts. With reference to existing work in the area, our approach is fully tool-equipped and relies on process algebraic encodings that support the definition and the use of *on-the-fly* algorithms for the adaptor generation.

Key-words: Components, Behavioral Interfaces, Mismatch, Composition, Adaptation, Process Algebra, LOTOS, On-the-fly Verification

A short version of this report is also available as "Behavioral Adaptation of Component Compositions based on Process Algebra Encodings", *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering ASE'2007 (Atlanta, Georgia, USA), November 2007.*

* radu.mateescu@inria.fr, INRIA / VASY project-team, Faculté des Sciences Mirande, bât. LE2I, 21078 Dijon, France

** pascal.poizat@inria.fr, INRIA / ARLES project-team and IBISC FRE 2873 CNRS, Tour Évry 2, 91000 Évry, France

*** salaun@lcc.uma.es, Dept. of Computer Science, Universidad de Málaga, Campus de Teatinos, 29071 Málaga, Spain

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot, 38334 Saint Ismier Cedex (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Adaptation comportementale des compositions de composants basée sur des encodages en algèbre de processus

Résumé : Le génie logiciel à base de composants (*Component-Based Software Engineering*, CBSE) et le calcul orienté service (*Service Oriented Computing*, SOC) favorisent la réutilisation de composants logiciels existants, respectivement de composants et services. Etant développées indépendamment, ces entités sont souvent incompatibles. L'adaptation logicielle a été proposée comme une solution à ce problème, avec l'objectif de produire, aussi automatiquement que possible, des adaptateurs, c.-à-d. des fragments de logiciel capables de résoudre les incompatibilités de manière non-intrusive. Nous proposons une approche pour la génération des protocoles d'adaptation à partir d'interfaces comportementales de composants et de contrats de composition. Comparée aux travaux existants dans le domaine, cette approche est complètement outillée et repose sur des encodages en algèbre de processus permettant la définition et l'utilisation d'algorithmes *à la volée* pour la génération d'adaptateurs.

Mots-clés : composants, interfaces comportementales, incompatibilité, composition, adaptation, algèbre de processus, LOTOS, vérification à la volée

1 Introduction

Components and services[1] can be developed separately by different third-parties. Their reusability is therefore often harmed by mismatch that appears at their different interface levels (signature, behavior/protocol, quality of service and semantics) [6]. Mismatch at the signature level is dealt with by state-of-the-art Interface Description Languages and middleware *e.g.*, CCM or .NET. The research has therefore jumped to the behavioral interface level where Behavioral Interface Description Languages (BIDL) and related formal verification techniques have emerged as a solution to detect behavioral mismatch, *e.g.*, [2, 8, 17, 18, 5]. However, once mismatch is detected, the issue of how components can be corrected to enable further composition arises. *Software Adaptation* [35, 28, 6] is a promising approach which aims at supporting software composition by generating as automatically as possible, pieces of software called *adaptors*. These are used to solve mismatch in a non-intrusive way, *i.e.*, without impacting on the components' code, which is impossible due to their black box nature.

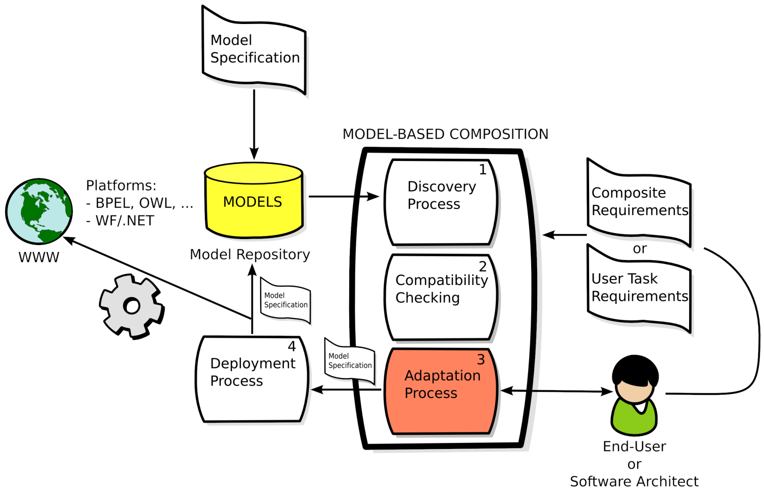


Figure 1: Research Context

Our works address composition in pervasive computing (see Figure 1), with the objective to compose automatically components to achieve end-user specified tasks at run-time or, more generally, to build new added-value composites. In this area, *behavioral adaptation* is particularly relevant to solve mismatch between the applicative level protocols of the components and hence to enable a larger number of compositions.

Model-based behavioral adaptation approaches rely on behavioral component descriptions and a *composition specification*, *i.e.*, an abstract level description of correspondences between interfaces that may help to solve mismatch out. These approaches can be divided into particular, restrictive and generative. *Particular approaches* [33] do not address the adaptation

[1] In the sequel, we use *component* as a general term covering both software components and services, *i.e.*, a software entity to be composed within a system.

from a general, automatable point of view, but propose specific practical solutions for particular situations instead. *Restrictive approaches* [23] try to solve the problem by cutting off (pruning) the behaviors that may lead to mismatch, thus restricting the functionality of the components involved. On the contrary, *generative approaches* [35, 11] try to accommodate the protocols without restricting the behavior of the components, by generating adaptors that act as mediators, storing and reordering messages and data when necessary, even at the risk of causing unexpected undesirable interactions. Recently, we have proposed a technique which combined the benefits of the last two approaches [14]. It can be considered as both generative and restrictive since we addressed behavioral adaptation by enabling message reordering, while we also removed incorrect behaviors.

The main issue in behavioral adaptation is to propose tools for the full automation of the proposed adaptation techniques [33, 23, 14]. Yet, the complexity of the adaptation process (which can be exponential, *e.g.*, when message reordering is enabled) is a problem that has to be solved before these techniques can be applied on pervasive systems. To solve this issue, we present in this report a new behavioral adaptation approach. Interfaces descriptions and composition specification are inherited from earlier work [14], however, the *adaptation technique* we rely on is new. Taking basis on the definition of an encoding of the adaptation constraints into the LOTOS [9] process algebra, we are then able to develop an adaptor construction technique where *on-the-fly* algorithms are used in place of ones working on a completely built state space, thus reducing the process complexity. Restrictive adaptation may restrict interactions between components in order to make the system work. This makes the assessment (evaluation) of adaptors and adapted systems a mandatory step of an adaptation approach in order to be sure they fulfil one's needs [28, 31]. An additional benefit of our LOTOS encoding is to support both the generation of adaptors and the assessment of adapted systems in a common framework.

This report is structured as follows. Section 2 introduces component behavioral interfaces and composition specifications. Section 3 presents our LOTOS encodings and Section 4 then describes the way adaptors can be obtained and assessed. Finally, Section 5 discusses related work and Section 6 ends with concluding remarks.

2 Components and Composition Specification

In this section we present our models for component interfaces and for the abstract description of requirements on the way they should interact.

2.1 Component Interfaces and Mismatch

Component interfaces are given using a signature and a behavioral interface. A *signature* is a set of operation profiles, *i.e.*, a disjoint union of provided operations and required operations. An operation profile is the name of an operation, together with its argument types, its return type and the exceptions it raises. In addition, we take into account *behavioral interfaces* through the use of *Labeled Transition Systems* (LTSs). LTSs favor user-friendliness and make the graphical specification of interfaces possible. Moreover, LTSs can be obtained from most formal behavioral specification languages (*e.g.*, process algebras [14] or workflow languages [16]). This makes LTSs an adequate formalism to represent behavioral interfaces in a model driven approach: other models can be transformed into LTSs prior to the composition and adaptation process we present in the sequel. The messages used in one component's LTS correspond to the operations of the component's signature and constitute what is called the component *alphabet*. Since we focus here on behavioral interoperability, we keep in signatures only the message names and we do not deal with operation arguments, return values or exceptions. Emissions are denoted with `!` and receptions with `?`.

Definition 1 (LTS) *A Labeled Transition System (*LTS*) is a tuple* (A, S, I, F, T) *where:* A *is an alphabet,* 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.*

Reuse of components is often harmed by mismatch, *i.e.*, from a behavioral point a view, cases such as:

- names of messages exchanged between components that do not correspond, a regular use of components making them interact on the same message names;
- ordering of messages that do not correspond;
- a message in a component that has no counterpart in another one, or that corresponds to more than one message;
- and more generally when the possible interactions between components may lead them to a blocking state.

Formally, this corresponds to the existence of *deadlock states* on the whole system viewed as a (global) LTS. For an LTS (A, S, I, F, T), a deadlock state is a state $s \in S$ which has no outgoing transition ($\nexists\,(s, l, s') \in T$) and is not final ($s \notin F$). Our approach aims at generating adaptors which will compensate such mismatch and make the components communicate properly.

2.2 Composition/Adaptation Specification

In this section we present *composition specifications*, an abstract way to denote composition and adaptation requirements. These requirements are later on put into practice by triggering the adaptor generation process. In order to help solving name mismatch, an important feature of composition specifications is first to support the explicit description of interactions between components. This is achieved thanks to *vectors* which relate the messages used in different components to implement some interaction. Vectors may involve any number of components and are not limited to interaction through shared message names (as in MSC, CSP and LOTOS) or complementary ones (as in CCS and the π-calculus) [4].

Definition 2 (Vector) *A* vector *for a set of components C_i, $i \in \{1,..,n\}$, is a tuple $\langle l_1, \ldots, l_n \rangle$ with $l_i \in A_i \cup \{\varepsilon\}$, where each A_i is the alphabet of component C_i and ε means that some component does not participate in the vector interaction.*

The prefixing of messages by component identifiers can be used in vectors in complement to ε omission in order to yield a digest notation, *e.g.,* a vector $\langle comm!, \varepsilon, comm? \rangle$ for components $\{c_1, c_2, c_3\}$ can be written $\langle c_1 \!:\! comm!, c_3 \!:\! comm? \rangle$.

Some situations require a specific application ordering of vectors to solve mismatch [14]. This can be adequately sorted out with a dynamic description of the way vectors should be applied, *e.g.* a regular expression as in [14] or more simply an LTS whose alphabet is a set of vectors. This *vector* LTS will be used as a way to denote composition specifications. Note that in cases where no specific ordering of vectors is required, the vector LTS can be reduced to a single state with a loop transition for each vector.

Definition 3 (Composition Specification) *A composition specification for a set of components C_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 an* LTS *whose alphabet is V.*

Message reordering is required when exchanged messages are not ordered correspondingly in communicating entities. A benefit of our approach is that this reordering has not to be made explicit in the writing of the composition specification to be taken into account in the generated adaptor. It is supported directly, if required, in our adaptor generation process (see an example of this on the registration prefix of Figure 6).

2.3 Approach Overview and Tool Support

The approach we propose for generating adaptors and assessing their effectiveness is illustrated in Figure 2, where the grey boxes indicate the newly developed tools: COMPOSITOR and SCRUTATOR.

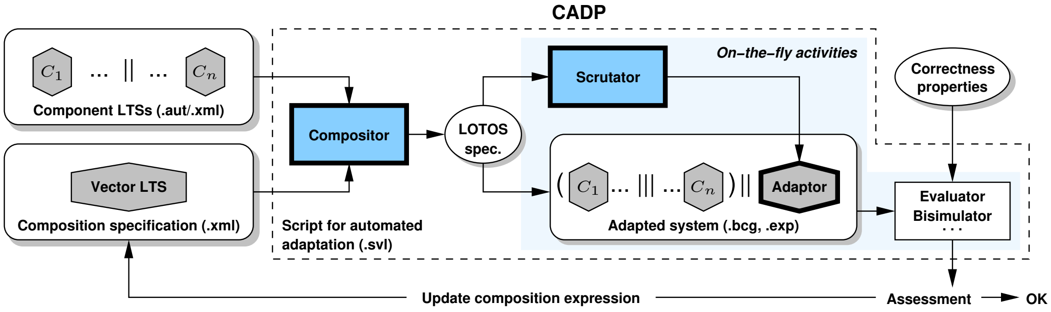


Figure 2: Overview of our Approach

The set of components interfaces and the composition specification are encoded into a LOTOS specification, which can be subsequently handled using the CADP toolbox [20]. This encoding is fully automated by COMPOSITOR. Supported inputs are XML and the `.aut` LTS textual format for component interfaces, and XML for composition specifications. The SCRUTATOR tool is used to explore the LTS of the LOTOS specification and to generate the LTS of the adaptor encoded in the `.bcg` compact LTS binary format. Additionally, SCRUTATOR is also able at the same time to reduce the adaptor LTS modulo various weak equivalence relations (τ-confluence, $\tau^*.a$, weak trace, etc.). With reference to existing adaptation techniques, SCRUTATOR is able to perform these three activities *on-the-fly*, without requiring the computation of the complete behavioral state space.

The adaptor LTS is then synchronized with the set of components, extracted from the LOTOS specification using an SVL script [20], in order to form the final system represented as an LTS network in the `.exp` format. The behavior of the final system can be explored using the SVL and EXP.OPEN [20] tools dedicated to the manipulation of LTS networks, and its correctness *w.r.t.* the properties required for adaptation can be verified with the various CADP tools (the model checker EVALUATOR, the equivalence checker BISIMULATOR, the graphical simulator OCIS, etc.). The results of the verification can trigger modifications of the composition specification, after which the whole cycle is repeated until all properties are fulfilled by the final system. COMPOSITOR and SCRUTATOR have been applied to generate adaptors for more than 100 examples (see Table 1 for some results).

2.4 Running Example: the eMuseum Service

The objective of the eMuseum service, illustrated in Figure 3, is to augment one's experience while visiting museums by supporting the display – on a personal portable device, *e.g.*

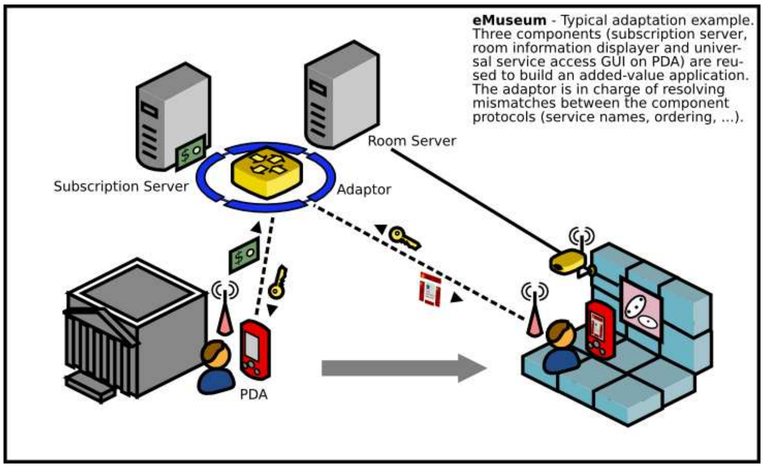


Figure 3: eMuseum adaptation case-study

a PDA or a smartphone – of information about seen pieces of art. This can be either video or text information, with video being sent only on a subscription all-in-package basis (while text information is for free). The service is built on top of three separately designed components whose behavioral interfaces are given in Figure 4 (final states are denoted with black circles). Component PDA can be used to register to one's environment's available (free or not) services – sending payment information, registering request and waiting for an identifier – and then use such services in a connected mode, based on a general query/reply schema, with reception of different multimedia formats. Component ROOM receives requests upon arrival in front of a painting and then sends related information, either text or video. It requires an identification for each request. Finally, component SUB handles registration and payment. It supports two modes: either guest mode or subscriber/user mode in which bank account is required to be debited.

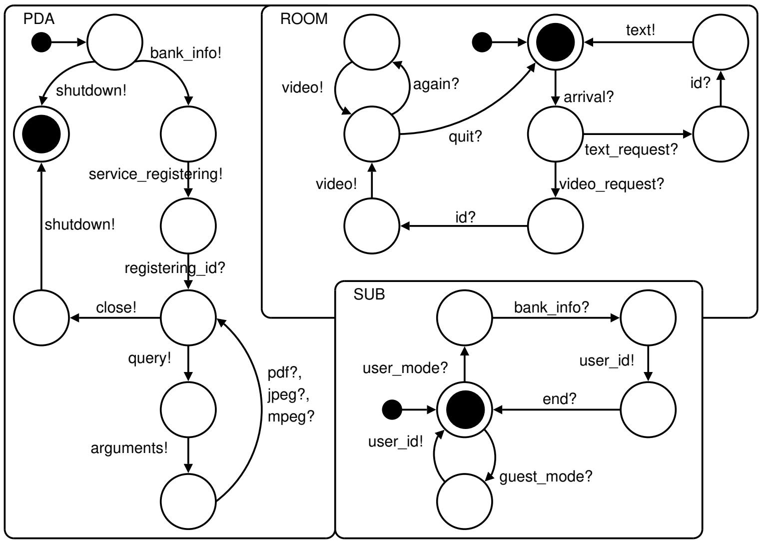


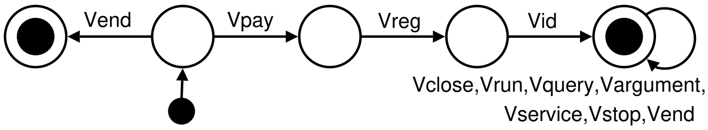
Figure 4: PDA, ROOM, and SUB Interfaces (LTSs)

As one may expect from the reuse of components designed by different third-parties, the three components cannot be composed as-is due to different kinds of mismatch. An example of name mismatch is between PDA sending service_registering and SUB waiting for either user_ or guest_mode. There are independent evolution problems as PDA may terminate with shutdown which has no counterpart in the other components. Message reordering is required to make the system work as for example payment information and registration requests are not done in the same order in PDA and SUB. Finally, the composition should also enforce more than deadlock freedom: subscription of PDA must be ensured before it accesses videos. In order to solve all these problems, different composition specifications can be given, for example depending whether one wants to make a subscriber or a guest contract. We describe below a composition specification on a subscriber contract assumption. In a guest contract, the sending of payment information by the PDA would for example be absorbed by the adaptor with a vector $\langle PDA\!:\!bank\_info!\rangle$.

The vectors are the following:

$$
\begin{aligned}
v_{end} &= \langle PDA\!:\!shutdown!\rangle \\
v_{pay} &= \langle PDA\!:\!bank\_info!, SUB\!:\!bank\_info?\rangle \\
v_{reg} &= \langle PDA\!:\!service\_registering!, SUB\!:\!user\_mode?\rangle \\
v_{id} &= \langle PDA\!:\!registering\_id?, SUB\!:\!user\_id!\rangle \\
v_{close} &= \langle PDA\!:\!close!, SUB\!:\!end?\rangle \\
v_{run} &= \langle ROOM\!:\!arrival?\rangle \\
v_{query} &= \langle PDA\!:\!query!, ROOM\!:\!video\_request?\rangle \\
v_{argument} &= \langle PDA\!:\!arguments!, ROOM\!:\!id?\rangle \\
v_{service} &= \langle PDA\!:\!mpeg?, ROOM\!:\!video!\rangle \\
v_{stop} &= \langle ROOM\!:\!quit!\rangle
\end{aligned}
$$

The related LTS is shown below:



3 Encoding into LOTOS

In this section we address the encoding of the systems' adaptation constraints into LOTOS. The objective of adaptation using such a constraint-based specification is to benefit from the specification language operational semantics to obtain an adaptor whose traces represent all possible (correct) interactions between components. The principle is to encode, and then compose in parallel, parts for:

- the components' LTSs, an adaptor must follow the components behaviors;
- the abstract requirements for composition and adaptation (*i.e.*, the composition specification), these represent the possibilities the adaptor has to translate messages (to deal with name mismatch) and to reorder them when needed;
- the desired system architecture – adaptor in-the-middle.

This encoding will enable (as described in Section 4) the automatic generation of adaptor protocols and the use of *on-the-fly* algorithms to increase, *w.r.t.* existing approaches, the efficiency of the adaptator generation and reduction process.

3.1 Introduction to LOTOS

We do a simplified presentation of behavioral specification in LOTOS. The reader may refer to [9] for a comprehensive introduction to this process algebra.

$$
\begin{array}{lll}
B ::= & \texttt{exit} & \textit{correct termination} \\
\mid & \texttt{stop} & \textit{deadlock} \\
\mid & g\,\texttt{;}B & \textit{action prefix} \\
\mid & B_1\,\texttt{[]}\,B_2 & \textit{choice} \\
\mid & B_1\,\texttt{|[}g_1,\ldots,g_n\texttt{]|}\,B_2 & \textit{parallel composition} \\
\mid & B_1\ \texttt{|||}\ B_2 & \textit{interleaving} \\
\mid & B_1\ \texttt{>>}\ B_2 & \textit{sequential composition} \\
\mid & \texttt{hide}\ g_1,\ldots,g_n\ \texttt{in}\ B & \textit{hiding} \\
\mid & P\,\texttt{[}g_1,\ldots,g_n\texttt{]} & \textit{process call}
\end{array}
$$

The *action*[2] *prefix* $g\texttt{;}B$ means that the execution of action g is followed by behavior B. The *choice* $B_1\,\texttt{[]}\,B_2$ denotes a behavior that may behave either as B_1 or as B_2. The *parallel composition* $B_1\,\texttt{|[}g_1,\ldots,g_n\texttt{]|}\,B_2$ means that behaviors B_1 and B_2 evolve in parallel, synchronizing on actions in the $g_1,\ldots,g_n$ list. The *interleaving* operator is a parallel composition in which behaviors evolve concurrently without synchronizing. The sequential composition B_1 `>>` B_2 executes behavior B_1 and behaves as B_2 upon B_1 termination (`exit`). The *hiding* construct `hide` $g_1,\ldots,g_n$ `in` B transforms the actions $g_1,\ldots,g_n$ of B into unobservable (τ) actions.

We want to distinguish emission from reception, yet `!` and `?` have a special meaning in LOTOS (supporting data transfer which is not used here). Therefore, we represent sent (resp. received) messages with a `_EM` (resp. `_REC`) suffix. Given an alphabet A, with $e! \in A$ and $r? \in A$, we define $enc(e!) = e\texttt{\_EM}$, $enc(r?) = r\texttt{\_REC}$, and $enc(A) = \{enc(l) \mid l \in A\}$.

3.2 Component LTS Encoding

Each state $s \in S$ of a component LTS c (A, S, I, F, T) is encoded as a LOTOS process c_s whose behavior is made up of as many branches as there are transitions outgoing from s. An additional branch is generated to model termination when s is final ($s \in F$). This is achieved using a specific `FINAL` action that will enable us to distinguish correct termination states (those with an incoming `FINAL` transition) from deadlocks.

```
process c_s [enc(A),FINAL] : func(c,s) :=
   enc(l_1); c_s_1 [enc(A),FINAL]
   [] ... []
   enc(l_m); c_s_m [enc(A),FINAL]
  [ [] FINAL; exit ]
endproc
```

where $A = \{l_1, \ldots, l_m, \ldots, l_n\}$, and $\{t \in T \mid \textit{source}(t) = s\} = \{(s, l_1, s_1), \ldots, (s, l_m, s_m)\}$ (*source* denotes the source state of a transition). Moreover, in LOTOS, a functionality has to be associated to each process declaration to indicate whether the process terminates

[2] An action in LOTOS may denote either an event (synchronizing on a gate) or a communication (emission/reception of messages).

(`exit`) or not (`noexit`). It is obtained as follows:

$$func(p,s) = \begin{cases} \texttt{exit} & \text{if } \exists s' \in F \text{ such that } reachable(c,s,s') \\ \\ \texttt{noexit} & \text{otherwise} \end{cases}$$

with

$$reachable(c,s,s') = \begin{cases} true & \text{if } \exists (s_1,\ldots,s_k) \in S^k, s_1 = s, s_k = s', \\ & \text{such that } \forall i \in \{1,\ldots,k-1\}\ \exists\ (s_i,t_i,s_{i+1}) \in T \\ false & \text{otherwise} \end{cases}$$

For deadlock states, the encoding is simply:

```
process c_s [enc(A),FINAL] : noexit := stop endproc
```

3.3 Composition Specification Encoding

A composition specification $CS = (V, L)$, with $L = (A_C, S_C, I_C, F_C, T_C)$ aims at coordinating the different components involved in a composition, and at working mismatch out. It is encoded by generating (i) a process for the vector LTS L, (ii) a process for each vector in V, and (iii) the interleaving of all these vector processes.

The correct ordering of vectors is ensured by the vector LTS process thanks to two actions for each vector v. A first one (`run_`v) is used to enable the corresponding vector process to receive messages. A second one (`rel_`v) releases the vector LTS and enables it to overlap vector applications, and hence reorder messages. The vector LTS process is encoded using a pattern similar to component LTSs in Section 3.2 (one process for each $s \in S_C$), yet with for each outgoing transition, the two specific actions `run_` and `rel_` used to trigger vector processes.

```
process LTS_s [A_L] : func(L,s) :=
   run_v1 ; rel_v1 ;LTS_s1 [A_L]
   [] ... []
   run_vm ; rel_vm ;LTS_sm [A_L]
  [ [] FINAL; exit ]
endproc
```

where $A_C = \{v_1, \ldots, v_n\}$, $A_L = \{\texttt{FINAL}\} \cup \bigcup_{v \in A_C} \{\texttt{run\_}v, \texttt{rel\_}v\}$, and $\{t \in T_C \mid \text{source}(t) = s\} = \{(s, v_1, s_1), \ldots, (s, v_m, s_m)\}$.

This encoding does not allow the vector LTS to overlap two instantiations of a given vector. In LOTOS, this can be specified by "`vector_`v = (... |||`vector_`v)" which means that `vector_`v may be launched as many times as necessary. However, this construct is forbidden by the CADP toolbox since it may build infinite systems. In the case one needs to design a composition expression in which a vector has to be launched several times with possible overlapping, the vector can be duplicated and identified with different names.

Vector processes communicate with components on shared actions. They have to receive all sent messages before beginning to emit some (*i.e.*, in $\langle c_1 : comm_1!, c_2 : comm_2!, c_3 : comm_3?\rangle$, $comm_1$ and $comm_2$ have to be received before the vector can send $comm_3$). There is no specific ordering between receptions (resp. between emissions) in a vector process. When a vector process executes a vector, it must be ready to interact with the component LTSs on the emissions, but next (after `rel_`v), the components' receptions can be postponed, and the vector LTS can launch another vector. This behavior is essential to make the reordering of messages possible. Each vector v is encoded as a process `vector_`v as follows:

```
process vector_v[Av] : exit:=
   (run_v;exit >> ( e1;exit||| ... |||ek;exit ) >>
    rel_v;exit >> ( r1;exit||| ... |||rm;exit ) >>
    vector_v [Av])
   [] FINAL; exit
endproc
```

where $A_v = \{\texttt{run\_}v, \texttt{rel\_}v, e_1, \ldots, e_k, r_1, \ldots, r_m, \texttt{FINAL}\}$, $\{e_1, \ldots, e_k\} = emissions(v)$, $\{r_1, \ldots, r_m\} = receptions(v)$, and functions *emissions* and *receptions* are defined as $emissions(\langle l_1, \ldots, l_n\rangle) = \{enc(l_i) \mid l_i = e!\}$ and $receptions(\langle l_1, \ldots, l_n\rangle) = \{enc(l_i) \mid l_i = r?\}$.

When a vector process executes a vector, it must be ready to interact with the component LTS on the emissions, but next (after `rel_`v), the components' receptions can be postponed, and the vector LTS can launch another vector. This behavior is essential to make the reordering of messages possible.

Finally, vector processes are interleaved, since they do not communicate altogether.

```
process vectors [AV] : exit :=
   (vector_v1 [Av1]||| ... |||vector_vn [Avn])
endproc
```

where $A_C = \{v_1, \ldots, v_n\}$, and $A_V = \bigcup_{v \in A_C} A_v$.

3.4 System Encoding

In this step, we generate a LOTOS expression corresponding to the whole system constraints from the LOTOS processes encoding the component LTSs, the ones encoding the composition specification, and respecting the desired system architecture (adaptor in-the-middle, intercepting all messages). This means that the component LTSs only interact together on `FINAL` (correct termination is when all components terminate) while they interact with vectors on actions used in their alphabets. The synchronizing between vector processes and vector LTS has been described earlier on. In addition, all actions that are not messages of the system, that is messages appearing in the involved components, are hidden as they represent internal actions of the adaptor we are building (*e.g.*, `run_` and `rel_` actions). They are the "mechanics" of adaptation and are not relevant for implementation. Such internal actions are inherent to adaptation via encoding (see, *e.g.*, τ actions corresponding to message transfer in the Petri net encoding of [14]). They will be removed by reduction steps of the adaptor generation process (see Section 4.1).

$$
\begin{array}{l}
\texttt{hide}\ A_L^*\ \texttt{in} \\
\quad (C_1\_I_{C_1}[enc(A_{C_1}),\texttt{FINAL}]\ \ |[\texttt{FINAL}]|\ldots|[\texttt{FINAL}]|\ \ C_n\_I_{C_n}[enc(A_{C_n}),\texttt{FINAL}]) \\
\qquad |[A_{CX},\texttt{FINAL}]| \\
\quad (\texttt{LTS}\_I_C[A_L]\ \ |[A_L]|\ \ \texttt{vectors}[A_V])
\end{array}
$$

where $A_L^* = A_L \backslash \{\texttt{FINAL}\}$ and $A_{CX} = \bigcup_{i \in \{1,\ldots,n\}} enc(A_{C_i})$.

3.5 Application to the eMuseum Service

For space reasons, we do not present the whole LOTOS encoding of our example ($\approx$ 400 lines), but instead we demonstrate the encoding of (part of) a component LTS on SUB and the encoding of vectors on v_{pay}. The SUB LTS has five states, which correspond in the encoding to five processes. One may note that, as the initial state (state 0) of SUB is final, the body of process `SUB_0` contains a choice with as second branch the action `FINAL` followed by a correct termination, `exit`. Other states of the components are declared locally to the process `SUB_0` using the `where` construct.

```
process SUB_0 [ASUB] : exit :=
   SUB_guest_mode_REC; SUB_1 [ASUB]
[] SUB_user_mode_REC; SUB_3 [ASUB]
[] FINAL; exit
where
   process SUB_1 [ASUB] : exit :=
      SUB_user_id_EM; SUB_0 [ASUB]
   endproc
[...]
endproc
```

where `ASUB` = `SUB_guest_mode_REC, SUB_user_mode_REC, SUB_user_id_EM, SUB_bank_info_REC, SUB_end_REC, FINAL`.

The vector v_{pay} is encoded as the LOTOS process `vector_Vpay`. This process is fired when the vector LTS interacts on action `run_Vpay`. The next interaction corresponds to the emissions, here there is a single one, namely `PDA_bank_info_EM`, carried out by component PDA. The vector LTS is the released (`rel_Vpay`) by `vector_Vpay` to enable it to execute other vectors. Here this will enable to begin vector v_{reg} and reorder bank information and registration. Receptions are then executed, here `SUB_bank_info_REC`, carried out by component SUB. Last, the process `vector_Vpay` is called again and may execute again the vector, or terminate (`FINAL`).

```
process vector_Vpay [AVPAY] : exit :=
( run_Vpay;exit >> (PDA_bank_info_EM;exit)  >>
  rel_Vpay;exit >> (SUB_bank_info_REC;exit) >>
  vector_Vpay [AVPAY])
[] FINAL;exit
endproc
```

where `AVPAY=run_Vpay, rel_Vpay, PDA_bank_info_EM, SUB_bank_info_REC, FINAL`.

4 Adaptor Generation

This section is devoted to the adaptor generation methodology. We first present the principle of our on-the-fly adaptor generation, then we show its application on the eMuseum service case-study and on several other examples from our database, and finally we illustrate how the generated adaptors can be verified by model checking in order to assess their adequacy w.r.t. the desired service expected from the adapted system.

4.1 Principle

An adaptor for a set of components is an LTS running in parallel with the component LTSs and guiding their execution in such a way that mismatches (deadlocks) are avoided and the ordering of messages imposed by the composition specification is guaranteed. Such an adaptor can be obtained from an LTS description of the whole system (components and composition specification) by keeping only the behaviors without mismatch, which amounts to cut the execution sequences leading to deadlock states. In the adaptation techniques that support such behavioral restriction, as [23, 14], the computation of the deadlock-free behaviors is done by performing a backward exploration of the explicit, *entirely constructed*, LTS by starting at the deadlock states and cutting all the transitions whose target state leads to a deadlock. Here we aim at more efficiency, avoiding the entire construction of the LTS, by generating the adaptor *on-the-fly* and exploring the LTS corresponding to the LOTOS specification of the whole system in a *forward* manner.

The CÆSAR compiler [22, 21] of CADP translates a LOTOS specification into a C program representing the corresponding LTS implicitly, by means of the successor function enumerating the outgoing transitions of a given state. This implicit LTS representation complies with the application programming interface defined by OPEN/CÆSAR [19], the generic environment of CADP dedicated to LTS exploration. Thus, a LOTOS encoding of the whole system allows to immediately benefit from all on-the-fly verification tools of CADP built using OPEN/CÆSAR. However, at the present time none of these tools provides the kind of LTS exploration needed for adaptor generation, and therefore we developed a new prototype tool, named SCRUTATOR, implementing this functionality. The adaptor generation procedure consists of two simultaneous activities, performed on-the-fly during a forward exploration of the LTS corresponding to the LOTOS specification of the whole system.

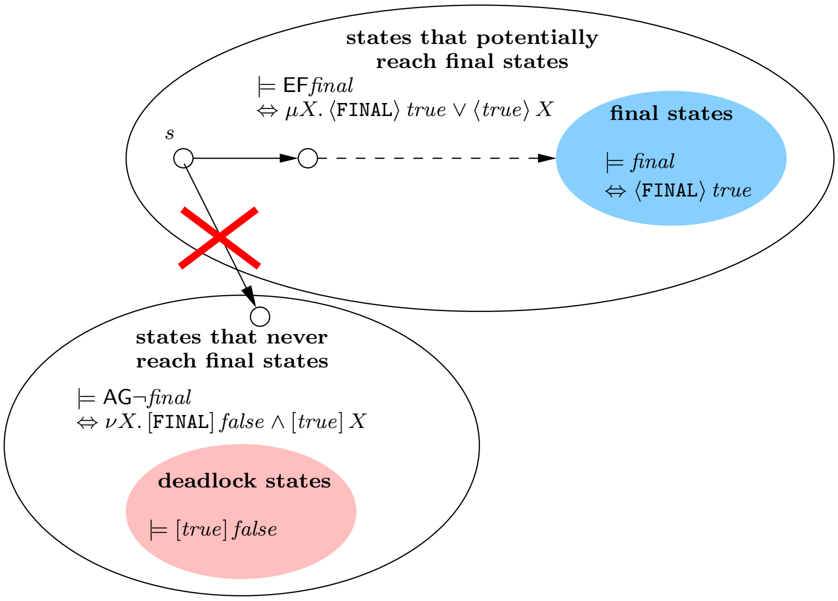


Figure 5: On-the-fly detection of states potentially leading to successful termination

Mismatch elimination. First, the execution sequences leading to mismatches (deadlocks) must be pruned. We do this by keeping, for each state encountered, only its successor states that potentially reach a successful termination state, which is source of a transition labeled with the action `FINAL`, as shown in Figure 5. Besides avoiding deadlocks (which are sink states reached by actions other than `FINAL`), this also avoids livelocks, *i.e.*, portions of the state space where some components get "trapped" and cannot reach their final states anymore. In a state-based setting based on Kripke structures, the desired successor states would satisfy the CTL [15] formula EF *final*, where the atomic proposition *final* characterizes successful termination states. In our action-based setting based on LTSs, those successor states would satisfy the μ-calculus [34] formula $\mu X. \langle \texttt{FINAL} \rangle\, true \vee \langle true \rangle\, X$ and they could be detected on-the-fly by invoking the EVALUATOR [27] model checker of CADP. However, this solution is not efficient since each invocation of EVALUATOR has a linear complexity *w.r.t.* the size of the LTS and therefore a sequence of invocations (in the worst case, one for each state of the LTS) may have a quadratic complexity.

The solution we adopted is to translate the evaluation of the formula into the resolution of the following boolean equation system (BES):

$$\{X_s =_\mu \bigvee_{s \xrightarrow{\texttt{FINAL}} s'} true \vee \bigvee_{s \rightarrow s''} X_{s''}\}$$

where a boolean variable X_s is true iff state s satisfies the propositional variable X corresponding to the μ-calculus formula. This BES (which is also used internally by EVALUATOR for checking the μ-calculus formula above) is solved on-the-fly using the algorithms of the CÆSAR_SOLVE library [26] of CADP, and particularly the algorithm dedicated to disjunctive

BESs (containing only $\vee$ operators in the right-hand sides of the equations), which stores in memory only the boolean variables (hence only the LTS states) and not the dependencies between them (and hence not the LTS transitions). The algorithms of CÆSAR_SOLVE are designed such that a sequence of resolutions has an overall linear complexity in the size of the BES; this is achieved by keeping the results of intermediate computations persistent between two subsequent resolutions. Thus, a state s potentially leading to a successful termination is detected by solving on-the-fly the variable X_s of the BES above; the overall complexity of the exploration enhanced with this detection remains linear *w.r.t.* the size of the LTS and does not store in memory the transitions, but only the states of the LTS.

Behavioral reduction. Second, the adaptor LTS obtained by pruning can be reduced on-the-fly modulo an appropriate equivalence relation in order to get rid of the internal actions and obtain an adaptor as small as possible. The current version of SCRUTATOR uses the algorithms in [25] to implement on-the-fly reductions modulo τ-confluence (a form of partial order reduction preserving branching bisimulation) and the $\tau^*.a$ and weak trace equivalences, both of which eliminate internal transitions and (for weak trace) determinize the adaptor LTS. We also plan to implement on-the-fly reductions for other equivalences, such as branching bisimulation; for the time being, the adaptors generated by SCRUTATOR can be reduced off-line modulo strong or branching bisimulation using the BCG_MIN tool of CADP.

The adaptation protocols described by our adaptors share message names and direction with the components. This is due to the process algebraic framework we rely on (LOTOS) where synchronizing is performed on the basis of shared actions. Note that in order to implement the adaptor protocol, one has to *mirror* all messages (reversing emissions and receptions, _EM and _REC) as the adaptor acts as an orchestrator in-the-middle of the components. This relabeling of adaptors is done using an SVL script.

4.2 Application to the eMuseum Service

Taking as input the LOTOS specification of the eMuseum service presented in Section 3, we applied SCRUTATOR in order to generate on-the-fly the corresponding adaptor, which has 410 states and 719 transitions without any reduction and 43 states and 63 transitions with τ-confluence coupled with weak trace reduction. The latter version can be further reduced modulo strong bisimulation using BCG_MIN (for efficiency reasons, the reductions performed on-the-fly do not attempt to produce the minimal LTS), yielding the adaptor with 31 states and 46 transitions shown in Figure 6.

In this adaptor, we stress the messages exchanged for registration. One may note that the adaptor mirrors the components' messages, follows the components' protocols, follows the orderings specified by vectors and vector LTS, and orders message reception/emission to avoid deadlocks. Here, the adaptor ensures PDA sends both bank_info and service_registering before they are transmitted (in the reverse order) as user_mode and bank_info to SUB which may thereafter reply with an identifier (user_id) that will be transmitted to PDA (registering_id).

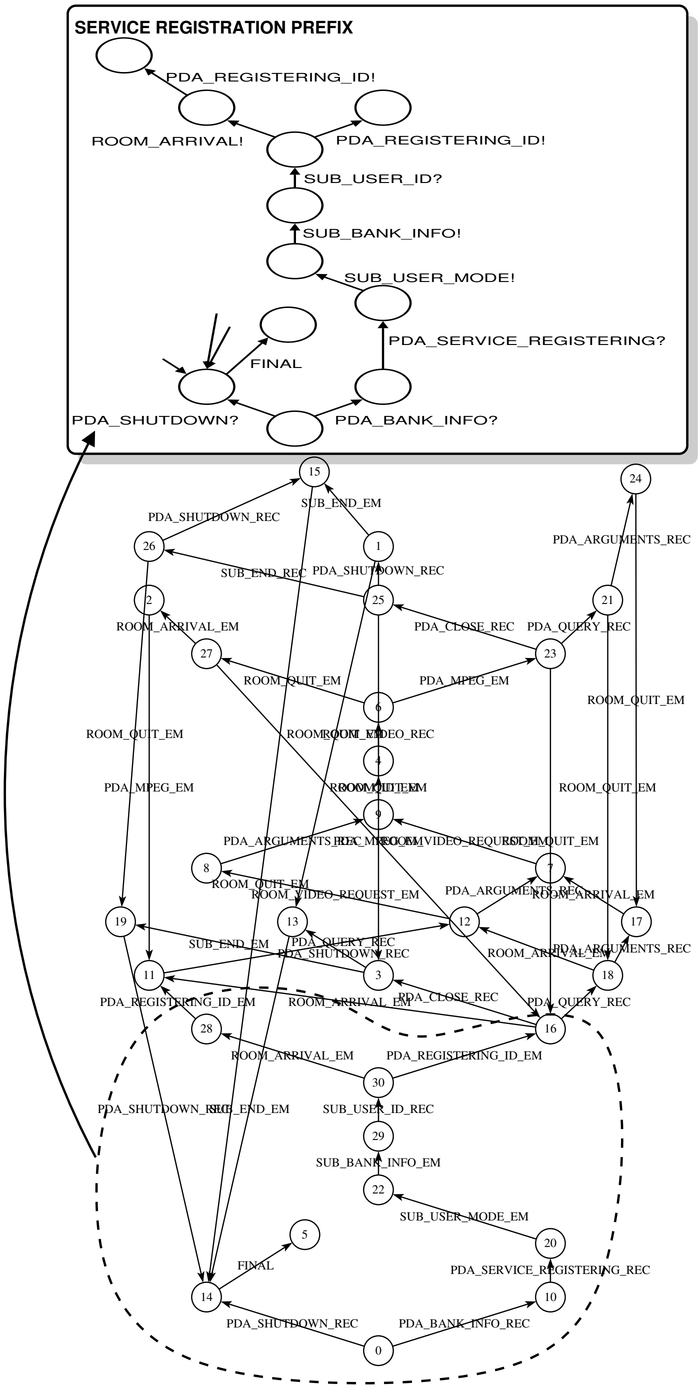


Figure 6: Adaptor LTS for the eMuseum service

Application	Adaptor LTS				LTS portion explored for reduced adaptor generation			
	raw		reduced					
	states	trans.	states	trans.	states	%	trans.	%
ebook	169,530	558,258	40,276	159,736	139,989	46.66	297,326	26.49
mail-system	12,273	31,877	148	398	12,048	84.98	7258	20.32
rate-service	599	982	54	81	240	33.76	197	15.02
emuseum (subscriber)	*410*	*719*	*43*	*63*	*771*	*89.76*	*728*	*48.05*
video-on-demand	229	406	12	15	18	64.29	17	47.22
sql-server	189	302	20	26	230	84.87	187	40.30
no-comm	151	215	24	43	156	78.39	109	39.64
emuseum (guest)	126	191	25	32	215	84.31	199	50.77
memory	94	147	35	53	74	78.72	70	47.62
pc-store	86	125	14	15	141	77.47	109	42.08
hi-fi	79	126	33	51	141	90.97	120	49.18
broadcast	66	127	34	63	49	74.24	45	34.88
library	59	68	21	21	186	93.47	163	60.15
deadlock	45	84	21	39	36	80.00	46	54.76
client-server	44	66	20	32	37	84.09	37	56.06
connected	41	57	14	17	31	75.61	40	70.18
flower	38	47	16	19	23	60.53	23	48.94
resource	23	28	10	10	10	43.48	10	35.71
restrictive	22	28	7	8	48	84.21	52	64.20
switch	14	16	7	7	7	50.00	6	37.50

Table 1: Examples of adaptor generation

Table 1 shows experimental measures on 20 examples from our database. For each experiment, the table gives the size of the "raw" adaptor LTS generated from the LOTOS specification by pruning deadlocks (columns 2, 3) and of the adaptor reduced on-the-fly by SCRUTATOR modulo weak trace equivalence combined with τ-confluence (columns 4, 5). Our running example of eMuseum service is referred to as "emuseum (subscriber)" in Table 1. The rightmost columns of the table indicate the portion of the LTS actually explored on-the-fly during the generation of the reduced adaptor, which can be significantly smaller (down to 33% of the states and 15% of the transitions) than the whole LTS. This illustrates the benefits of on-the-fly adaptation *w.r.t.* the approach based on explicit construction of LTSs employed in [24, 14]. Moreover, we observe that the ratio between the number of transitions explored and the transitions of the LTS is smaller than the ratio concerning the states. This aspect — which becomes more important with the increase of the branching factor of the LTS, proportional to the number of components in the adapted system — further penalizes the explicit approach *w.r.t.* the on-the-fly approach, because the former requires to store all LTS transitions in memory in order to compute the adaptor, whereas the latter requires to store only LTS states. The largest example given in Table 1 ("ebook") took a little more than one minute of computation on a 731 MHz, 512 MB PC running Linux.

4.3 Verification of the Adaptation

The use of Lotos to encode not only the system components but also the composition specification has the additional advantage of allowing to verify the correctness of the adaptor alone or of the final system (components and adaptor) using the Cadp tools. Verification is needed in order to ensure that the whole system is correct *w.r.t.* the desired service to be obtained after adaptation: indeed, the user may provide a wrong composition specification, which does not solve all the mismatches existing between the components, and could lead to an erroneous adaptor. For the eMuseum service example, one can use Evaluator to check temporal properties expressed in regular alternation-free μ-calculus. This has been done on ten properties, as those illustrated below.

Safety. The client cannot get a video before it has successfully subscribed (sending its bank information):

$$[(\neg\texttt{SUB\_BANK\_INFO\_EM})^*.\texttt{PDA\_MPEG\_EM}]\,\mathit{false}$$

Liveness. Every request sent by the client will be eventually answered by the service:

$$[\mathit{true}^*.\texttt{PDA\_QUERY\_REC}]\,\mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg\texttt{ROOM\_VIDEO\_REC}]\,X$$

The safety property holds on the final system obtained after adaptation but not on the PDA alone, since it involves an action of SUB which has no counterpart in PDA; the correct ordering of these actions has been ensured by adaptation on an adequate composition specification. The liveness property holds on the final system, but fails[3] on the system before adaptation (made of components and vector Lts) because of mismatches. The shortest counterexample sequence (16 transitions) produced by Evaluator corresponds to the following chain of vectors: v_{pay} (emission of banking information by PDA), v_{reg} (emission of registering demand by PDA), v_{id} (emission of user id by SUB). This chain is then followed by a request $\mathit{PDA}{:}\mathit{query}!$ (captured by vector v_{query}) sent by the PDA before the vector v_{run} was executed; since the ROOM component is not ready to accept a request, this leads to a deadlock.

The whole machinery of translation into Lotos, on-the-fly generation of the adaptor, and verification of properties was automated using Svl scripts, as indicated in Figure 2. We give below the Svl script handling all operations from the on-the-fly generation of the adaptor for the eMuseum example to the verification of the final system obtained after adaptation.

[3] When checking the properties on the system before adaptation, one should use the original action names, before the mirroring of the adaptor. Thus, the liveness property checked on the original system is $[\mathit{true}^*.\texttt{PDA\_QUERY\_EM}]\,\mu X.\langle \mathit{true}\rangle \mathit{true} \wedge [\neg\texttt{ROOM\_VIDEO\_EM}]\,X$.

```
% DEFAULT_LOTOS_FILE="emuseum.lotos"

(* On-the-fly generation and reduction of the adaptor *)

% caesar.open "emuseum.lotos" ./scrutator.a -tauconfluence -weaktrace \
%             -potential FINAL "emuseum_scr.bcg" ;

(* Mirroring of adaptor's actions and strong minimization *)

"emuseum_adaptor.bcg" = strong reduction of
    total rename "\(.*\)_XXX" -> "\1_REC" in
    total rename "\(.*\)_REC" -> "\1_EM"  in
    total rename "\(.*\)_EM"  -> "\1_XXX" in
        "emuseum_scr.bcg" ;

(* Generation of the components *)

"room_0.bcg" = strong reduction of room_0 ;
"pda_0.bcg" = strong reduction of pda_0 ;
"sub_0.bcg" = strong reduction of sub_0 ;

(* Generation of the adapted system *)

"emuseum_final.bcg" = strong reduction of
    ("room_0.bcg" |[FINAL]| "pda_0.bcg" |[FINAL]| "sub_0.bcg")
        |[room_quit_REC, room_video_EM,
          room_again_REC, room_arrival_REC,
          room_text_EM, room_id_REC,
          room_video_request_REC, room_text_request_REC,
          pda_pdf_REC, pda_mpeg_REC,
          pda_query_EM, pda_arguments_EM,
          pda_service_registering_EM, pda_registering_id_REC,
          pda_jpeg_REC, pda_bank_info_EM, pda_close_EM,
          pda_shutdown_EM, sub_stat_REC,
          sub_end_REC, sub_guest_mode_REC,
          sub_user_mode_REC, sub_user_id_EM,
          sub_bank_info_REC, FINAL]|
    "emuseum_adaptor.bcg" ;

(* Verification of the adapted system *)

"diag_safe.bcg" = verify "safety.mcl"   in "emuseum_final.bcg" ;
"diag_live.bcg" = verify "liveness.mcl" in "emuseum_final.bcg" ;
```

5 Related Work

The problem of automatically adapting components to foster reuse is not a new discipline. In [30, 28], the authors propose a framework with adaptation features. They are able to retrieve and compose automatically a set of components supporting parts of the system desired properties and adapt them in order to fulfill a given specification. They use axiomatic Rosetta specifications and rely on theorem-proving techniques as the underlying formal tool support. We do not support component retrieval for the moment. Moreover, as our objective is to address behavioral composition, we rather use behavioral specifications and related tools.

Several recent adaptation proposals [35, 33, 23, 11, 14] also focus on solving behavioral mismatch between components. Brogi et al. (BBCP) [10, 11] present a methodology for generative behavioral adaptation where component behaviors are specified with a subset of the π-calculus and composition specifications with name correspondences. An adaptor generation algorithm is used to refine the given specification into a concrete adaptor which is able to accommodate both message name and protocol mismatch. This approach has recently been used to obtain adaptor implementations for services [12]. Inverardi et al. (IT) [24, 23] address the enforcement of behavioral properties out of a set of components. Starting from the specification with MSCs of the components to be assembled and of LTL properties (liveness or safety) 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. They follow a restrictive adaptation approach, hence are not for example able to reorder messages when required. More recently, in [14], we have proposed an automated adaptation approach that was both generative and restrictive, and supports adaptation policies and system properties described by means of regular expressions of vectors. It superseded both IT (as it supported message reordering) and BBCP (which could generate dumb adaptors [10] and has no tool-support), yet it built on algorithms based on synchronous products and Petri nets encodings with a resulting exponential complexity for the computation of adaptors.

All these works suffer from the cost of computing global adaptors for the whole set of components. Some work have recently tried to address this issue either at the computation time – by supporting the incremental computation of adaptors [31] – or at deployment time – by distributing the adaptors over the components [3]. Yet, they build on algorithms for the computation of global adaptors previously defined by their authors. What we have defined here is a technique which enables the one-the-fly removal of incorrect interactions and behavioral reduction of adaptors. As such, this technique can be used in restrictive (or generative and restrictive) approaches, or when the adaptor size is too big and requires reduction before implementation. Compared to the existing adaptation approaches, ours also benefits from being based on a formal toolset which has been intensively validated on large case studies over the last years, namely CADP. In particular, compared to our own previous approaches [14, 31], the one at hand deals with the same mismatch but follows a simplified process and has a better complexity. Using COMPOSITOR to get a LOTOS specification, one can directly rely on CADP and our external module for it, SCRUTATOR to get adaptors. Last, the algorithm dealing with reordering presented in [14] suffered from state explosion in some cases – *e.g.* in case of components with emission looping behaviors (due to message storing). This issue was related to the Petri net encoding used, and is avoided in the current solution which has passed examples failing with [14].

Software adaptation can be related to controller synthesis [32], which focuses on the generation of controllers with respect to a given system (called plant) designed as a finite automaton and properties to be ensured. However, both approaches are fundamentally different because controller synthesis goal is to influence (when possible) the behavior of the controlled system, while adaptation works on black-box components and should be non intrusive. In addition, controller synthesis is a restrictive approach whereas efficient adaptation approaches are also generative.

The fact that adaptation can be restrictive, or may produce non envisioned interaction scenarios using reordering, makes assessment an important issue in software adaptation. Verification of adapted systems is tackled in [29] where Atelier B is used to check properties of adaptors completely specified with the B method. We believe behavioral equivalences or model checking are more adequate than theorem proving for the verification of behavioral adaptors. A benefit of the approach we present here is that adaptor generation and adaptor assessment are tool-supported in a common framework, Lotos and Cadp. Amongst the numerous component or service verification approaches, we may cite one, also based on LOTOS, for Fractal components [5] which could be used in complement with our adaptation approach to assess systems.

To sum up, compared to the related work introduced above, our approach relies on different strengths which are: a simple yet expressive mapping language, a self-contained and simple approach to generate adaptors, the reuse when possible of reliable and optimized tools that have been widely validated and used, on-the-fly algorithms to generate adaptor protocols, automation of the approach thanks to prototype tools we have implemented, and possible verification (assessment) of adapted systems in the same framework as the adaptor generation.

6 Concluding Remarks

Software adaptation is a promising area of Software Engineering to solve mismatch between components and hence increase their reusability level. Results on model-based adaptation are now mature enough to be put into practice on implementation frameworks [24, 12, 16], without requiring the need of specific adaptive middleware [1], but rather use directly the features provided by the target framework, which is more compatible with an MDE approach.

In this report, we have proposed a completely automated approach for the generation of adaptor protocols which supersedes state-of-the-art work on two important elements. First, the use of a process algebra encoding, namely Lotos, supports both the adaptor generation process and its assessment/verification in a common and unified framework, Cadp. Moreover, this enabled the definition and the use of on-the-fly techniques for the pruning of component interactions leading to deadlocks and for the reduction of adaptors' size. This is particularly interesting in contexts where adaptor size is relevant: adaptation at run-time or in service composition for pervasive and ambient intelligence systems [7] where the end-user is equipped with low-resources portable devices. Both encoding and adaptor generation are fully automatized thanks to the CADP toolbox and the tools we have developed, COMPOSITOR and SCRUTATOR.

Perspectives are within the context of the framework for component and service automatic composition and adaptation presented in the introduction. A first perspective aims at avoiding the complete construction of adaptors at design or deployment time, and rather construct them while interacting with running components or services. However, while composing the components at run-time, one should avoid to engage into execution branches that may lead to deadlock situations. A first step towards run-time adaptation is presented in [13] where a bounded forward-looking simulation algorithm is used to ensure deadlock freedom on the chosen bound limits. To ensure full correctness, we could reuse the encoding and on-the-fly technique presented here, provided we extend one CADP simulation tool (OCIS) with on-the-fly verification. A second perspective concerns the tool integration of our model-based works, as presented here, with our implementation related works [16].

References

[1] Special Issue on Adaptive Middleware. *Communications of the ACM*, 45(6):30–64, 2002.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, 1997.

[3] M. Autili, M. Flammini, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis of Concurrent and Distributed Adaptors for Component-based Systems. In *Proc. of EWSA'06*, volume 4344 of *LNCS*, pages 17–32. Springer-Verlag, 2006.

[4] T. Barros, R. Boulifa, and E. Madelaine. Parameterized models for distributed java objects. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2004 (Madrid, Spain)*, volume 3235 of *LNCS*, pages 43–60. Springer-Verlag, September 2004.

[5] T. Barros, A. Cansado, and E. Madelaine. Model-Checking Distributed Components: the Vercors Platform. In *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software FACS'2006 (Prague, Czech Republic)*, ENTCS. Elsevier, 2006.

[6] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, chapter Towards an Engineering Approach to Component Adaptation, pages 193–215. Springer-Verlag, 2006.

[7] S. Ben Mokhtar, Liu J., N. Georgantas, and V. Issarny. QOS-aware Dynamic Service Composition in Ambient Intelligence Environments. In *Proc. of ASE'05*, pages 317–320. ACM Press, 2005.

[8] M. Bernardo and P. Inverardi, editors. *Formal Methods for Software Architectures*, volume 2804 of *LNCS*. Springer-Verlag, 2003.

[9] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

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

[11] A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 63(1):39–56, 2006.

[12] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 27–39. Springer-Verlag, 2006.

[13] J. Camara, G. Salaün, and C. Canal. Run-time Composition of Mismatching Transactions. In *Proc. of SEFM'07*, 2007.

[14] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*, pages 63–77. Springer-Verlag, 2006.

[15] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[16] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. Relating Model-Based Adaptation and Implementation Platforms: A Case Study with WF/.NET 3.0. In *Proc. of WCOP'07*, 2007.

[17] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of ASE'03*, pages 152–163. IEEE Computer Society, 2003.

[18] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*. ACM Press, 2004.

[19] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 68–84. Springer-Verlag, 1998.

[20] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–162. Springer-Verlag, 2007.

[21] H. Garavel and W. Serwe. State space reduction for process algebra specifications. *Theoretical Computer Science*, 351(2):131–145, February 2006.

[22] H. Garavel and J. Sifakis. Compilation and verification of lotos specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[23] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. Synthesis of Correct and Distributed Adaptors for Component-based Systems: an Automatic Approach. In *Proc. of ASE'05*, pages 405–409. ACM Press, 2005.

[24] P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.

[25] R. Mateescu. On-the-fly state space reductions for weak equivalences. In Tiziana Margaria and Mieke Massink, editors, *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems FMICS'05 (Lisbon, Portugal)*, pages 80–89. ERCIM, ACM Computer Society Press, September 2005.

[26] R. Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. *STTT Journal*, 8(1):37–56, 2006.

[27] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[28] B. Morel and P. Alexander. Automating Component Adaptation for Reuse. In *Proc. of ASE'03*, pages 142–151. IEEE Computer Society, 2003.

[29] I. Mouakher, A. Lanoix, and J. Souquieres. Component Adaptation: Specification and Verification. In *Proc. of WCOP'06*, 2006.

[30] J. Penix and P. Alexander. Efficient specification-based component retrieval. In *Proc. of ASE'99*, pages 139–170. Kluwer Academic Publishers, 1999.

[31] P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*, pages 141–156. Springer-Verlag, 2007.

[32] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proc. of the IEEE*, 77(1):81–98, 1989.

[33] R. H. Reussner. Automatic component protocol adaptation with the CoConut/J tool suite. *Future Generation Computer Systems*, 19(1):627–639, 2003.

[34] C. Stirling. *Modal and Temporal Properties of Processes*. Springer-Verlag, 2001.

[35] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

Centre de recherche INRIA Grenoble – Rhône-Alpes
Inovallée, 655, avenue de l'Europe, Montbonnot - 38334 Saint Ismier Cedex (France)

Centre de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes, 4, rue Jacques Monod - Bât. G - 91893 Orsay Cedex (France)
Centre de recherche INRIA Nancy – Grand Est : 615, rue du Jardin Botanique - 54600 Villers-lès-Nancy (France)
Centre de recherche INRIA Rennes – Bretagne Atlantique : Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399