Adaptation of Open Component-based Systems

Pascal Poizat[1,2] and Gwen Salaün[3]

[1] IBISC FRE 2873 CNRS – University of Evry Val d'Essonne, France
[2] ARLES project, INRIA Rocquencourt, France
pascal.poizat@inria.fr
[3] Department of Computer Science, University of Málaga, Spain
salaun@lcc.uma.es

Abstract. Software adaptation aims at generating software pieces called adaptors to compensate interface and behavioural mismatch between components or services. This is crucial to foster reuse. So far, adaptation techniques have proceeded by computing *global adaptors for closed systems* made up of a fixed set of components. This is not satisfactory when the systems may evolve, with components entering or leaving it at any time, *e.g.*, for pervasive computing. To enable adaptation on such systems, we propose tool-equipped adaptation techniques for the computation of *open systems adaptors*. Our proposal also support *incremental adaptation* to avoid the computation of global adaptors.

1 Introduction

Compared to hardware components, software components (or services) are seldom reusable *as is* due to possible mismatch that may appear at different levels [10]: signature, behaviour, quality of service and semantics. Once detected, mismatch has to be corrected. However, it is not possible to impact directly on the components source due to their black-box nature. *Software adaptation* [27, 10] aims at generating, as automatically as possible, pieces of software called *adaptors* which are used to solve mismatch in a *non intrusive* way. To this purpose, model-based adaptation techniques base upon behavioural component interface descriptions and abstract properties of the adaptation called *adaptation contracts* or *mappings*. Dedicated middleware [2] can be used to put the adaptation process into action once an adaptor model (or implementation) has been obtained, but this is out of scope here.

Existing (global) adaptation approaches [27, 21, 14, 8, 11] proceed by generating a *global adaptor* for the whole system which is seen as a *closed* one. First of all this is costly. Moreover, when a component uses a service which does not relate through mapping to the other components' services, then either its use is prevented by adaptation (to avoid deadlock), or it is made internal (related events sent by the component are absorbed by the adaptor). Taking into account that new components, and hence new services, may be available in the future is not possible. Global adaptation approaches suffer from the fact that the adaptor has to be computed each time something changes in the system and are therefore not efficient in contexts such as pervasive systems [20], where services

are not fixed or known from scratch. They may evolve, *e.g.*, depending on the mobility of the user –moving around, different services are discovered and may be used, or on connectivity or management issues –some services may be temporary or definitely unavailable. In this paper we address these issues by extending a previous work for the adaptation of closed systems [11] in order to support (i) the *adaptation of open systems* and accordingly, (ii) an *incremental process for the integration and adaptation of open software components*. The definitions and algorithms we present have been implemented in Adaptor [1], our tool for model-based adaptation.

The paper is structured as follows. Section 2 presents our open systems component model and our adaptation techniques for such systems. In Section 3, we present the incremental adaptation of open systems, addressing the addition and the removal of components. Incremental adaptation has an added value at design-time, where the integration of components is known to be a difficult task, which gets worse when components have not been designed altogether from the beginning and therefore when adaptation connectors are needed. This typical use of the incremental adaptation of open systems is illustrated in Section 4. We end with comparison of related work and concluding remarks.

2 Open Systems Adaptation

In this section we address the adaptation of open systems. We first recall a formal model for basic sequential components originating from [11]. Then we define an open composition model on top of it thanks to the definition of (i) compositional vectors, (ii) open synchronous product and (iii) open component-based systems and their semantics. In a second step, our adaptation algorithms are presented.

2.1 Components

Alphabets, the basis for interaction, correspond to an event-based signature. An alphabet A is a set of service names, divided in provided services, $A^?$ (elements denoted as $e?$), required services, $A^!$ (elements denoted as $e!$) and internal actions (denoted with τ). The mirror operation on an alphabet element is defined as $\overline{e?} = e!$, $\overline{e!} = e?$, and $\overline{\tau} = \tau$. Moreover, for an alphabet A, $\overline{A} = \{\overline{e} \mid e \in A\}$.

Component interfaces are given using a signature and a behavioural interface. A *signature* is a set of operation profiles as in usual component IDLs. This set is a disjoint union of *provided* operations and *required* operations. *Behavioural interfaces* are described in a concise way using a sequential process algebra, sequential CCS: `P ::= 0 | a?.P | a!.P | τ.P | P1+P2 | A`, where `0` denotes termination, `a?.P` (resp. `a!.P`) a process which receives `a` (resp. sends it) and then behaves as `P`, `τ.P` a process which evolves with the internal action τ (also denoted using `tau` in figures) and behaves as `P`, `P1+P2` a process that acts either as `P1` or `P2`, and `A` the call to a process defined by an equation `A = P`, enabling recursion. The CSS notation is extended using tags to support the definition of initial (`[i]`) and final states (`[f]`). `0` and `0[f]` are equivalent. In order to define adaptation algorithms, we use the process algebra operational semantics

to retrieve *Labelled Transition Systems* (LTS) from the interfaces, *i.e.*, tuples $\langle A, S, I, F, T\rangle$ where A is the alphabet (set of communication events), S is the set of states, $I \in S$ is the initial state, $F \subseteq S$ is the set of final states, and $T \subseteq S \times A \times S$ are the transitions. The alphabet of a component LTS is built on this component's signature. This means that for each provided operation p in the signature, there is an element $p?$ in the alphabet, and for each required operation r, an element $r!$.

2.2 Open Component Systems

Vectors are an expressive mechanism to denote communication and express correspondences between events in different processes. In this work vectors are extended to take into account open systems and keep track of their structuring. For this purpose, vectors are defined with reference to an (external) alphabet which relates component events to composite systems external interfaces (see Defs. 3 and 4, below).

Definition 1 ((Compositional) Vector). *A compositional vector (or vector for short) v for a set of LTSs $L_i = \langle A_i, S_i, I_i, F_i, T_i\rangle, i \in \{1, \ldots, n\}$ and an (external) alphabet A_{ext} is an element of $A_{\text{ext}} \times (A_1 \cup \{\varepsilon\}) \times \ldots \times (A_n \cup \{\varepsilon\})$. Such a vector is denoted $e : \langle l_1, \ldots, l_n\rangle$ where $e \in A_{\text{ext}}$ and for every i in $\{1, \ldots, n\}$, $l_i \in A_i \cup \{\varepsilon\}$. ε is used in vectors to denote a component which does not participate in a communication.*

The definition of an open synchronous product yields a tree-shaped structure for labels which makes it possible to keep trace of the structuring of composite components. When needed we may restrict to the observable part of labels, defined as $\text{obs}(e : \langle l_1, \ldots, l_n\rangle) = e$. Moreover, labels of simple LTS can be related to composite ones using $l : \langle l\rangle$ for any label l.

Definition 2 (Open Synchronous Product). *The open synchronous product of n LTSs $L_i, i \in \{1, \ldots, n\}$ with reference to a set of vectors V (defined over these LTSs and an external alphabet A_{ext}) is the LTS $\Pi((L_1, \ldots, L_n), A_{\text{ext}}, V) = \langle A, S, I, F, T\rangle$ such that: $A = A_{\text{ext}} \times A_1 \times \ldots \times A_n$, $S = S_1 \times \ldots \times S_n$, $I = (I_1, \ldots, I_n)$, $F = F_1 \times \ldots \times F_n$, and T contains a transition $((s_1, \ldots, s_n), e : \langle a_1, \ldots, a_n\rangle, (s'_1, \ldots, s'_n))$ iff there is a state $(s_1, \ldots, s_n)$ in S, there is a vector $e : \langle l_1, \ldots, l_n\rangle$ in V and for every i in $\{1, \ldots, n\}$:*

- *if $l_i = \varepsilon$ then $s'_i = s_i$ and $a_i = \varepsilon$,*
- *otherwise there is a transition (s_i, a_i, s'_i) with $\text{obs}(a_i) = l_i$ in T_i.*

Remark. In practice, we reduce S (resp. F) to elements of S (resp. F) which are reachable from I using T.

Example 1. Let us suppose we have two LTSs, L_1 and L_2, with one transition each: $(I_1, a?, S_1)$ for L_1 and $(I_2, b!, S_2)$ for L_2. Different sets of vectors may express different communication semantics:

- $\{\tau : \langle a?, b!\rangle\}$ (services $a?$ and $b!$ being synchronised) will produce a product LTS with a single transition: $((I_1, I_2), \tau : \langle a?, b!\rangle, (S_1, S_2))$;

– $\{a? : \langle a?, \varepsilon\rangle\ ,\ b! : \langle \varepsilon, b!\rangle\}$ (services $a?$ and $b!$ left open to the environment) will produce a product LTS with four transitions:
$((I_1, I_2), a? : \langle a?, \varepsilon\rangle, (S_1, I_2))$, $((I_1, I_2), b! : \langle \varepsilon, b!\rangle, (I_1, S_2))$,
$((S_1, I_2), b! : \langle \varepsilon, b!\rangle, (S_1, S_2))$, $((I_1, S_2), a? : \langle a?, \varepsilon\rangle, (S_1, S_2))$.

If we take this second case into account and make a product with an LTS L_3 with two transitions, $(I_3, c!, S_3)$ and $(S_3, d?, S'_3)$, and vectors $\{\tau : \langle a?, c!\rangle\ ,\ \tau : \langle b!, d?\rangle\}$, we get a product LTS with two transitions:
$(((I_1, I_2), I_3), \tau : \langle a? : \langle a?, \varepsilon\rangle, c!\rangle, ((S_1, I_2), S_3))$,
$(((S_1, I_2), S_3), \tau : \langle b! : \langle \varepsilon, b!\rangle, d?\rangle, ((S_1, S_2), S'_3))$.

Composites denote sets of hierarchical connected open components.

Definition 3 (Composite (or Open Component System)). *A composite is a tuple* $\langle C, A_{\text{ext}}, B_{\text{int}}, B_{\text{ext}}\rangle$ *where:*

- C *is a set of component instances, i.e., an* Id*-indexed set of LTS* $L_i, i \in Id$ *(*Id *usually corresponds to the integers* $\{1, \ldots, n\}$*),*
- A_{ext} *is an (external) alphabet,*
- B_{int} *is a set of vectors, with each vector* $e : \langle l_1, \ldots, l_n\rangle$ *in* B_{int} *being such that* $e = \tau$*, there is some* i *in* $\{1, \ldots, n\}$ *such that* $l_i \neq \varepsilon$ *and there is at most one* j *in* $\{1, \ldots, n\}\backslash\{i\}$ *such that* $l_j \neq \varepsilon$*.* B_{int} *denotes internal (hidden) bindings between the composite sub-components, When clear from the context, such vectors can be denoted as couples* (l_i, l_j)*,*
- B_{ext} *is a set of vectors, with each vector* $e : \langle l_1, \ldots, l_n\rangle$ *in* B_{ext} *being such that* $e \neq \tau$*, there is some* i *in* $\{1, \ldots, n\}$ *such that* $l_i \neq \varepsilon$*, and for every* k *in* $\{1, \ldots, n\}\backslash\{i\}$*,* $l_k = \varepsilon$*.* B_{ext} *denotes external bindings between the composite sub-components and the composite interface itself. When clear from the context, such vectors can be denoted as couples* (e, l_i)*.*

Our structure of composites supports (through model transformation) existing hierarchical ADLs such as the Fractal one [9] or UML 2.0 component diagrams [17]. Note that with reference to these models we have an exact correspondence between their notions of component ports and component interfaces in what we call alphabets. Our model for bindings is more expressive than the Fractal ADL or UML 2.0 ones as we enable bindings between services with different names, which is mandatory to support adaptation.

Example 2. Let us take a component system described in an ADL (Fig. 1) where a batch processing client interacts with a database server to perform SQL requests. Our graphical notation is inspired from Fractal ADL, yet a textual notation is also supported. This model can be transformed into the following composite structure:

⟨ {Client,SQLServer}, {launch?,exitCode!},
{(log!,id?), (request!,sqlQuery?), (reply?,sqlValues!), (reply?,sqlError!), (end!, ε)},
{(launch?,run?), (exitCode!,exitCode!)} ⟩.

Open synchronous product is used to give a formal semantics to composites.

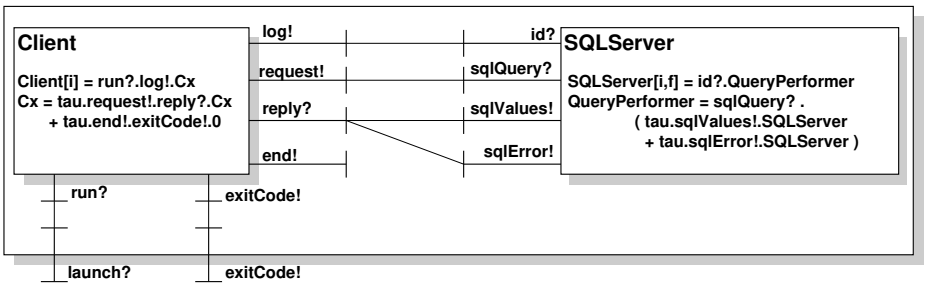

Fig. 1. The Client and SQLServer Example Architecture.

Definition 4 (Composite Semantics). *The semantics of a composite* $\mathcal{C} = \langle C, A_{\text{ext}}, B_{\text{int}}, B_{\text{ext}} \rangle$ *is the LTS* $L(\mathcal{C}) = \Pi(C, A_{\text{ext}}, \mathcal{V})$ *with* $\mathcal{V} = B_{\text{int}} \cup B_{\text{ext}} \cup B_\tau$ *where* $B_\tau = \bigcup_{i \in \{1,\ldots,n\}} \{b_{\tau,i}\}$ *and* $b_{\tau,i} = \tau : \langle l_1, \ldots, l_n \rangle$ *where* $l_i = \tau$ *and* $l_k = \varepsilon$ *for every* k *in* $\{1, \ldots, n\} \backslash \{i\}$.

In presence of several hierarchical levels (composites of composites), composite sub-components are first translated into LTSs using their semantics (Def. 4), *e.g.*, the semantics of a composite $\mathcal{C} = \langle \{C_1, \ldots, C_n\}, A_{\text{ext}}, B_{\text{int}}, B_{\text{ext}} \rangle$ where some C_i is a composite can be obtained replacing C_i by $L(C_i)$ in $\mathcal{C}$.

2.3 Mismatch and Mappings

Composition correctness is defined in the literature [10] either at the composition model level – using deadlock freedom – or at the components' protocols level – using compatibility or substitutability notions. As we want to use compositions in finding ways to correct mismatching components, we rely on the first approach. States without outgoing transitions are legal if they correspond to final states of the composed components. Therefore, we define *deadlock* (and hence mismatch) for a composite with a semantics $\langle A, S, I, F, T \rangle$ as a state $s \in S$ of the composition which has no outgoing transition ($\nexists (s, l, s') \in T$) and is not final ($s \notin F$). A deadlock is a state of the composition in which respective component protocols are incompatible, due to signature and/or behavioural mismatch. Note that if the former one can be solved using correspondences and renaming, the latter one requires more subtle techniques. This is also the case when correspondences evolve over time (*e.g.*, in Example 4 below, id? in SQLServer corresponds first to log! in Client and later on to nothing).

Example 3. Let us get back to the composite presented in Example 2. Mismatch in the example is due first to mismatching names. Moreover, even with an agreement on the service names, the fact that the client works in a connected mode (sending its log only once and disconnecting with `end`) while the server works in a non connected mode (requiring an id at each request) will also lead to behavioural mismatch after the first request of the client has been processed by the server.

We propose regular expressions of open vectors as the means to express adaptation contracts. A regular expression (or regex for short) over some basic domain $\mathcal{D}$ is the set of all terms build on: d (ATOM), $R1.R2$ (SEQUENCE), $R1 + R2$ (CHOICE), $R1*$ (ITERATION) and N (USE), with $d \in \mathcal{D}$, $R1$ and $R2$ being regular expressions, and N being an identifier referring to a regex definition $N = R$. Such definitions can be used to structure regex but we forbid recursive definitions for operational reasons.

Definition 5 ((Adaptation) Mapping). *An adaptation mapping (or mapping for short) for a composite $\mathcal{C} = \langle C, A_{\text{ext}}, B_{\text{int}}, B_{\text{ext}} \rangle$ is a couple (V, R) where V is a set of (compositional) vectors for the LTSs in C and A_{ext}, and R is a regular expression over V.*

Example 4. To work our system out, one easily guesses that the client has first to be launched, to connect, the system then runs for some time and finally the client disconnects and exits. This is specified for example using the mapping $\mathsf{M}=\mathsf{v}_{\mathsf{launch}}.\mathsf{v}_{\mathsf{cx}}.\mathsf{M}_{\mathsf{run}}.\mathsf{v}_{\mathsf{dx}}.\mathsf{v}_{\mathsf{exit}}$ with vectors $\mathsf{v}_{\mathsf{launch}}$=launch?:<run?, ε >, v_{cx}= τ :<log!,id?>, v_{dx}= τ :<end!, ε >, and $\mathsf{v}_{\mathsf{exit}}$=exitCode!:<exitCode!, ε >. Yet, it is more complicated to know what should be done while the system runs ($\mathsf{M}_{\mathsf{run}}$), excepted of course that events are exchanged for requests/results and that somehow a reset (resending the client identification) should be used. Therefore, one may choose to keep this part of the mapping abstract: $\mathsf{M}_{\mathsf{run}}$= $(\mathsf{v}_{\mathsf{req}}+\mathsf{v}_{\mathsf{res}}+\mathsf{v}_{\mathsf{err}}+\mathsf{v}_{\mathsf{reset}})^*$ with vectors v_{req}= τ :<request!,sqlQuery?>, v_{res}= τ :<reply?,sqlValues!>, v_{err}= τ :<reply?,sqlError!>, and $\mathsf{v}_{\text{reset}}$= τ :< ε,id?>.

Discussion on the mapping notation. Mappings are made up of the definition of possible correspondences (vectors) and a dynamic description over such correspondences (regex). Several behavioural languages may be used to this purpose. We have presented regex for their simplicity. However, the only requirement for the algorithms presented in Section 2.4 to work is to be able to obtain from the mapping an LTS where transitions are labelled by vectors (Algorithm 1, line 13). Currently, Adaptor supports both regex and the direct use of LTS. Message Sequence Charts (MSC) where arrows are labelled by vectors are a user-friendly alternative. LTS can be obtained from MSC using, *e.g.*, [24]. We are also investigating the use of techniques from the composition of Web services [6] in order to get automatically possible correspondences between services (vectors) and ease the user task in the context of end-user composition in pervasive systems.

2.4 Algorithms

Using a mapping and component behavioural interfaces, an adaptor can be generated automatically for a closed system following results from, *e.g.*, [21, 14, 8, 11]. Here, our algorithms (Alg. 1 and 2) enable adaptation on open systems. Algorithm 1, works by translating into a Petri net [16] the constraints of a correct adaptor. This choice is done as Petri nets enable to see messages exchanged between components as resources of the adaptor, to de-synchronise messages and

Algorithm 1 `build_PetriNet`

inputs mapping M, components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$
outputs Petri net $\mathcal{N}$

1: $\mathcal{N}$:= `empty_PetriNet()` *// all remaining actions operate 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 `[i@s_j]` **end for**
4: put a token in place `[i@I_i]` *// I_i is the initial state of C_i*
5: **for all** $a! \in A_i$ **do** add a place `??a` **end for**
6: **for all** $a? \in A_i$ **do** add a place `!!a` **end for**
7: **for all** $(s, e, s') \in T_i$ with $l = \text{obs}(e)$ **do**
8: add a transition with label $\overline{l}$, an arc from place `[i@s]` to the transition and an arc from the transition to place `[i@s']`
9: **if** `l` has the form $a!$ **then** add an arc from the transition to place `??a` **end if**
10: **if** `l` has the form $a?$ **then** add an arc from place `!!a` to the transition **end if**
11: **end for**
12: **end for**
13: $L_R = (A_R, S_R, I_R, F_R, T_R)$:= `get_LTS_from_regex`(R) *// see [13]*
14: **for all** $s_R \in S_R$ **do** add a place `[R@s_R]` **end for**
15: put a token in place `[R@I_R]` *// I_R is the initial state of L_R*
16: **for all** $t_R = (s_R, e : \langle e_1, \ldots, e_n \rangle, s'_R) \in T_R$ with $\forall i \in \{1, \ldots, n\}\ l_i = \text{obs}(e_i)$ **do**
17: add a transition with label `e`, an arc from place `[R@s_R]` to the transition and an arc from the transition to place `[R@s'_R]`
18: **for all** l_i **do**
19: **if** l_i has the form $a!$ **then** add an arc from place `??a` to the transition **end if**
20: **if** l_i has the form $a?$ **then** add an arc from the transition to place `!!a` **end if**
21: **end for**
22: **end for**
23: **for all** $(f_r, f_1, \ldots, f_n) \in F_R \times F_1 \times \ldots \times F_n$ **do**
24: add a (loop) `accept` transition with arcs from and to each of the tuple elements
25: **end for**
26: **return** $\mathcal{N}$

Algorithm 2 `build_adaptor`

inputs mapping M, components $C_1, \ldots, C_n$ with each $C_i = \langle A_i, S_i, I_i, F_i, T_i \rangle$
outputs adaptor $Ad = \langle A, S, I, F, T \rangle$

1: $\mathcal{N}$:= `build_PetriNet`$(M, \{C_1, \ldots, C_n\})$ *// see Algorithm 1*
2: **if** `bounded`$(\mathcal{N})$ **then** L := `get_marking_graph`$(\mathcal{N})$
3: **else** L := `add_guards`(`get_cover_graph`$(\mathcal{N})$) **end if**
4: Ad :=`reduction`(`remove_paths_to_dead_states`(L))
5: **return** Ad

therefore support reordering when required. The encoded adaptor constraints are as follows. First, the adaptor must mirror each component interface (places and transitions are generated from component interfaces, lines 2–12). It must also respect the adaptation contract specified in the mapping (places and transitions are generated, lines 14–22, from an LTS description of the mapping obtained in line 13). Algorithm 2 works out the building of the adaptor from this

net using several functions. `bounded` checks if a Petri net is bounded. If so, its marking graph is finite and can be computed (`get_marking_graph`); if not, then we rely on an abstraction of it, a cover graph (`get_cover_graph`), where the ω symbol abstracts any token number > 0. Due to the over-approximation of cover graphs, `add_guards` is used on them to add a guard `[#??a>1]` (`#??a` meaning the number of tokens in place `??a`) on any `a!` transition leaving a state where `#??a` is ω. `remove_paths_to_dead_states` recursively removes transitions and states yielding deadlocks. The optimising of resulting adaptors is achieved thanks to reduction techniques (`reduction`). Branching reduction [25] is the most appropriate choice as it does not require a strict matching of τ transitions like strong equivalence. In addition, branching equivalence is the strongest of the weak equivalences, therefore properties restricted to visible actions (*e.g.,* deadlock freedom, but also safety and fair liveness) are preserved by reduction modulo branching equivalence.

Example 5. We present in Figure 2 the Petri net generated for the Client and SQLServer example. To help the reader, we present separately the different parts of the net which are generated for Client (top left), SQLServer (top right) and the mapping (bottom left). The accept transition and the dashed places are used to glue the three subnets. The resulting adaptor is also shown (bottom right). It is more complex than its contract, which demonstrates the need for automatic adaptation processes as presented here.

Our algorithms are supported by Adaptor which relies on ETS [18] for open product computation, TINA [7] for the marking and cover graph computation, and on CADP [12] for adaptor reduction. Due to the computation of marking/cover graphs for the Petri net encodings, this algorithm is in theory exponential in the size of the Petri net, which in turn is related to the sum of the sizes of the component protocols and their alphabets ($\sum_{i\in\{1,\ldots,n+1\}}(|S_i|+|A_i|)$). Yet, in practice, the adapted components are sequential, hence parts of generated Petri nets are 1-bounded which lowers the complexity. The incremental mechanism for adaptation we present in the next section also helps in minimising the complexity of computing adaptors.

3 Incremental Adaptation of Open Component Systems

We may now describe an incremental adaptation approach suitable to open systems. At design-time, it helps in the design and integration of component-based systems, grounding on automatic adaptor-connector generation. At run-time, it avoids the computation of global adaptors and supports evolving systems.

3.1 Architectural Style

The definition of an architectural style is the support for the description, reasoning and implementation of software architectures. As far as design-time incremental adaptation is concerned, resulting design architectural models will

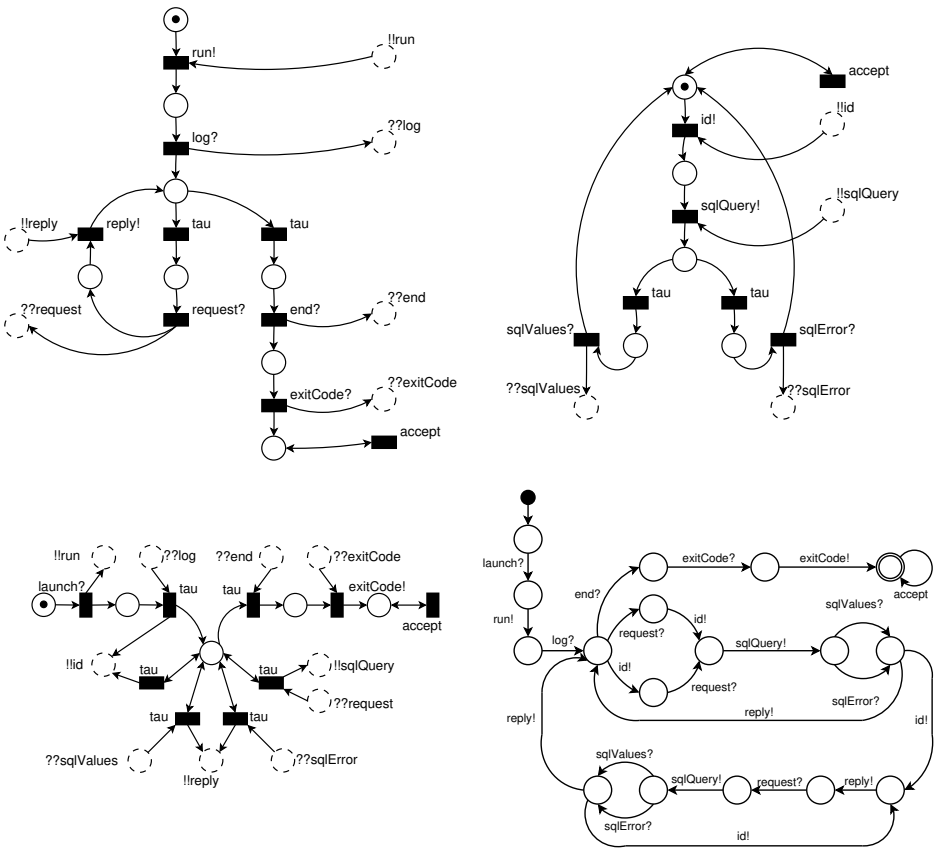

Fig. 2. The Client and Server Example Adaptor Generation.

respect our style. As far as run-time incremental adaptation is concerned, communication mechanisms are constrained by it.

Two kinds of entities are distinguished: components and adaptors. Components implement the system's functionalities or services. Adaptors are used as intermediates to avoid deadlock and enforce different coordination policies whose properties are specified in an abstract way in mappings. A component is only connected to its adaptor, and interacts with the rest of the system through it. If the component does not require adaptation, our approach automatically generates a *no-op* adaptor which reproduces from an external point of view exactly the same behaviour as the component. Adaptors can be connected to other adaptors in order to ensure the system's global correctness.

To support implementation or run-time adaptation, two kinds of interactions have to be distinguished at an adaptor level: with its component and with other adaptors. Adaptors have to agree on a common implementation communication protocol to communicate altogether. Mismatch between components which has

been solved thanks to adaptors should not be transferred to a mismatch between adaptors which should communicate correctly by construction. *Prefixing* will help there. Communications with the environment are prefixed by the component identifier and communications with other adaptors are prefixed by the identifier of the components these adaptors are in charge of. As far as communication between adaptors and components is concerned, communications are not prefixed as adaptation should be transparent for the adapted component. The use of prefixing is demonstrated in Section 4 on our application.

3.2 Assessment

Adaptors may impose service restriction due to the application of the function removing paths to dead states in the adaptation algorithm (Alg. 2, line 4). These are hard to detect by hand and assessment procedures are therefore required to help the designer (for design-time adaptation) or the end-user (for run-time adaptation). We propose tool-supported procedures based on alphabet comparison and property checking.

Alphabet-based assessment and comparisons may be used either to check the adapted system for services or more specifically to compare the adapted component with reference to the original one. In the first case we may check either successfully synchronised services (obtained hiding in LTSs any transition $e :< l_1, \ldots, l_n >$ where there is only one i such that $l_i \neq \varepsilon$) or actions left open to the environment, possibly new services provided by composites (obtained hiding in LTSs any transition $e :< l_1, \ldots, l_n >$ where there are at least two different $l_i \neq \varepsilon$). Comparison between original and adapted components can be performed on the same basis (internal or external comparison) through difference between their respective alphabets.

Property checking is a finer grained technique and may efficiently be used to detect more subtle architectural flaws. Classical properties such as liveness properties (*e.g.*, *any request will eventually be satisfied*, see Sect. 4 for an application of this) can be easily formalised reusing patterns [15], and then checked against the adapted system model (LTS) using model-checkers. An interesting benefit is that, when the property is unsatisfied, model-checkers return back a counter-example sequence of service calls that may help modifying mappings.

3.3 Addition and Suppression of Components

In this section, we present the algorithms for the addition and for the suppression of components. In the addition algorithm (Alg. 3), a component (C_{n+1}) is adapted and integrated into an existing composite (possibly empty). Adaptation is performed using only the component to be added, a given mapping and adaptors of components referred to in the mapping. Assessment is used to check the result of the integration. It is important to note that, as a preliminary step, automatically built mappings can be proposed. When the system is empty, a no-op mapping, $\left(V, (\sum_{v \in V} v)*\right)$ with $V = \{(C_{n+1}{:}e, e) \,|\, e \in A_{C_{n+1}}\}$, simply wraps the added component. When there are already components to

Algorithm 3 `addition`

inputs composite $\mathcal{C} = \langle \{C_1, AC_1, \ldots, C_n, AC_n\}, A_{ext}, B_{int}, B_{ext} \rangle$, component C_{n+1}
output new composite $\mathcal{C}^a = \langle \{C_1, AC_1, \ldots, C_n, AC_n, C_{n+1}, AC_{n+1}\}, A^a_{ext}, B^a_{int}, B^a_{ext} \rangle$

repeat
  $M :=$ `get_mapping`()// *designer or end-user given*
  $AC_{n+1} :=$`build_adaptor` (M, `get_cn'ed_adaptors_from_mapping`(M)$\cup\{C_{n+1}\}$)
  $B'_{ext} :=$ `get_externals_from_mapping`(M)
  $A^a_{ext} := A_{ext} \cup \{e \,|\, (e, e') \in B'_{ext}\}$
  $B^a_{int} := B_{int} \cup$ `get_internals_from_mapping`(M)
  $B^a_{ext} := B_{ext} \cup B'_{ext}$
  $\mathcal{C}^a := \langle \{C_1, AC_1, \ldots, C_n, AC_n, C_{n+1}, AC_{n+1}\}, A^a_{ext}, B^a_{int}, B^a_{ext} \rangle$
until `assess_or_stop`($\mathcal{C}^a$)// *human-interaction may stop the process*
return $\mathcal{C}^a$

Algorithm 4 `suppression`

inputs composite $\mathcal{C} = \langle \{C_1, AC_1, \ldots, C_n, AC_n\}, A_{ext}, B_{int}, B_{ext} \rangle$, $C_{k,k\in\{1,\ldots,n\}}$
output new composite $\mathcal{C}'$

$\{C_1, \ldots, C_m\} :=$ `reachable`(C_k,$\mathcal{C}$,`added_after`($C_k, \mathcal{C}$))
$\mathcal{C}' :=$ `build_composite`(`added_before`($C_k, \mathcal{C}$))
for all $C_{i,i\in\{1,\ldots,m\}}$ **do** $\mathcal{C}' :=$ `addition`($\mathcal{C}', C_i$) **end for**
return $\mathcal{C}'$

communicate with in the system, a trivial mapping, $\big(V, (\sum_{v\in V} v)*\big)$ with $V = \{(C_i{:}e, \overline{e}) \,|\, C_i{:}e \in A_{\text{ext}} \wedge \overline{e} \in A_{C_{n+1}}\}$, can be tested. In this algorithm, function `get_cn'ed_adaptors_from_mapping` iterates over the set of vectors V of the mapping M. For each v in V, if v respects the form given for B_{int} in Definition 3, we can obtain a couple (l_i, l_j) and then, looking at the n adaptors alphabets, determine the adaptor l_i corresponds to. Function `get_externals_from_mapping` (resp. `get_internals_from_mapping`) returns the set of couples (e, l_i) (resp. (l_i, l_j)) from the vectors $e : \langle l_1, \ldots, l_n \rangle$ of the mapping M that respect the form given for B_{ext} (resp. B_{int}) in Definition 3. The suppression algorithm (Alg. 4) first computes all the components that have been added after the component to be removed, and are reachable (in terms of the architectural graph topology) from it. The suppression may impact all these components, therefore their corresponding adaptors are successively updated if needed using the component addition algorithm. In this algorithm we use the following functions. Function `added_after` (resp. `_before`) returns the ordered set of all components of the composite $\mathcal{C}$ added after (resp. before) the component C_k. Function `reachable` returns all components of the composite $\mathcal{C}$ present in a given filtering set (`added_after` results) which are reachable from the component C_k. Finally, `build_composite` is used to build a composite applying the addition algorithm on an ordered set of components (result of `added_before` in the algorithm), and reusing mappings from the former composite construction. Mappings are therefore kept with adaptors while building the system. Removing a component induces the suppression of its adaptor, but also the update of all the adaptors interacting with it. In

the worst case, this corresponds to recompute all adaptors which is as costly as the regular case in global adaptation approaches where the adaptor is always recomputed.

4 Application

We have validated our approach on several examples: the dining philosopher problem, a video-on-demand system, a pervasive music player system, and several versions of a library management application. We present here a simplified version of the latter one. The system manages loans in a library. Components were chosen non recursive (this corresponds to the notion of transactional services) to obtain readable resulting adaptors.

The first component, `LIB`, tests if a book is available in the library or has been borrowed by a user.

```
LIB[i,f] = isBorrowed?. (available!.0 + borrowed!.0)
```

A no-op adaptor, `ALIB`, is first computed using a no-op mapping generated automatically as presented in Section 3.3. Then, a second component, `SUB`, is added. It is used as a front-end to the `LIB` component and checks if a user is a subscriber of the library. If not, `SUB` replies with the `notAvailable!` message, otherwise it tests if the requested book is borrowed or available.

```
SUB[i,f] = info?.isRegistered?.(isBorrowed!.SUB_AUX + notAvailable!.0)
SUB_AUX  = (notBorrowed?.available!.0 + borrowed?.notAvailable!.0)
```

It is obvious that the components present both name and protocol mismatch, therefore the trivial mapping fails assessment. We recall that events are prefixed except for those corresponding to interactions between the adaptor and its component (see Section 3.1). To work the mismatch out, a mapping `M1=(v1.v2.(v3+v4.(v5.v6+v7.v3)))*` is proposed, with vectors

```
v1 = SUB:info?         : <LIB:ε, info?>
v2 = SUB:isRegistered? : <LIB:ε, isRegistered?>
v3 = SUB:notAvailable! : <LIB:ε, notAvailable!>
v4 = τ                 : <LIB:isBorrowed?, isBorrowed!>
v5 = τ                 : <LIB:available!, notBorrowed?>
v6 = SUB:available!    : <LIB:ε, available!>
v7 = τ                 : <LIB:borrowed!, borrowed?>
```

In Figure 3 we present the architecture resulting from our incremental integration and adaptation process. The left hand part is related to the architecture after the addition of `SUB` and its adaptor, `ASUB`. The overall figure corresponds to the final architecture (after all components have been added, see `BOR` below). The architecture is computed automatically using Algorithm 3.

It was not possible to give all binding names (A_{ext}, B_{int}, B_{ext}) in the figure due to lack of place. However, bindings here are between ports of same name as the architectures are correct by construction using adaptation. The adaptor `ASUB` generated from `M1` is shown in Figure 4.

A third component, `BOR`, receives requests for loans and checks if the book can be borrowed or not (`id!` stands for identifiers of the user and book).

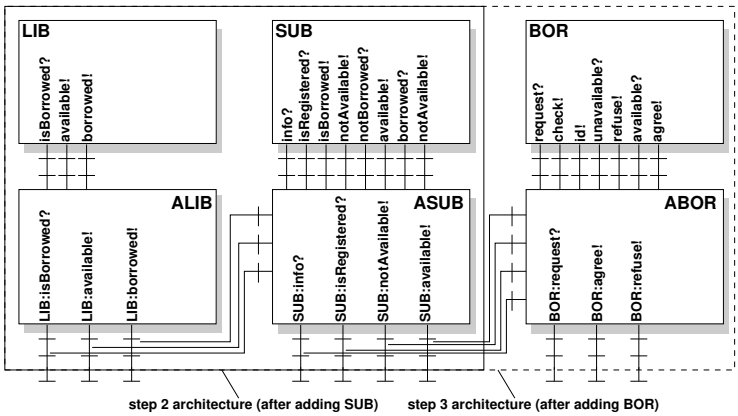

Fig. 3. The Library Example Architecture.

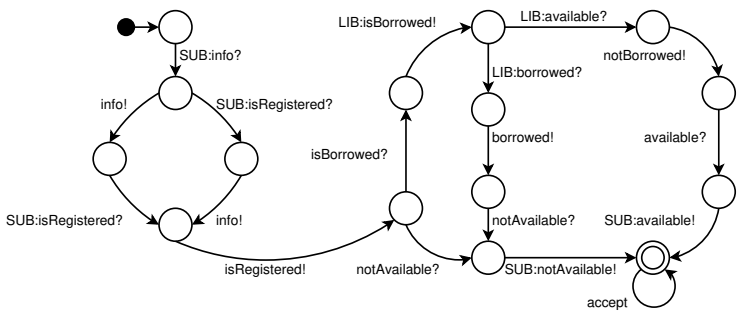

Fig. 4. The Library Example Adaptor.

```
BOR[i,f] = request?.check!.id!.
             ( unavailable?.refuse!.0 + available?.agree!.0 )
```

Component `BOR` can be connected to component `SUB` using the mapping `M2` to make all three components work together. Note in the mapping below that reordering is needed since `BOR` sends first the `check!` message and then information about the request `id!`, whereas `SUB` accepts first `info?`, and then the request message `isRegistered?`. Therefore, the following sequence belongs to the `ABOR` adaptor: `check?.id?.SUB:info!.SUB:isRegistered!`.

```
M2 = v1.v2.v3.(v4.v5+v6.v7)*

v1 = BOR:request?:<SUB:ε,request?>  v5 = BOR:refuse!:<SUB:ε,refuse!>
v2 = τ:<SUB:isRegistered?,check!>   v6 = τ:<SUB:available!,available?>
v3 = τ:<SUB:info?,id!>              v7 = BOR:agree!:<SUB:ε,agree!>
v4 = τ:<SUB:notAvailable!,unavailable?>
```

Now, let us illustrate assessment procedures on the system made up of the three components, and their corresponding adaptors. This system is quite simple

(33 states, 60 transitions, and 25 labels) since the adaptation process has removed all incorrect interactions. External alphabet contains messages `BOR:request?`, `BOR:agree!`, `BOR:refuse!`, and also all the messages left observable in the previous steps, *i.e.,* `LIB:isBorrowed?`, `LIB:available!`, etc. The synchronised alphabet contains all the remaining messages which are connected internally. In addition, the system is deadlock-free and verifies the following liveness property:

```
[true*](["BOR:request?"]
  (mu X. (<true> true and [not ("BOR:agree!" or "BOR:refuse!")]X)))
```

It states that messages `BOR:request!` are always followed after a finite number of steps either by a message `BOR:agree?` or `BOR:refuse?`. Basically, this means that all requests are always replied, which corresponds to the classic pattern "AG request⇒EF reply" encoded in μ-calculus. This property was automatically checked using Evaluator the model-checker of CADP. Consequently, the `BOR` adaptor is validated, and the final correct architecture is as presented earlier on.

Let us now remove component `SUB`. This can be done for update purposes or just because the loan check is simplified not to take into account that the user has to be a subscriber. `BOR` is the only connected component added after `SUB`. A new mapping is given for component `BOR` to connect it directly to component `LIB`, `M2' = v1.v2.v3.(v4.v5+v6.v7)*` with vectors

```
v1 = BOR:request?:<LIB:ε,request?>  v5 = BOR:refuse!:<LIB:ε,refuse!>
v2 = τ:<LIB:isBorrowed?,check!>     v6 = τ:<LIB:available!,available?>
v3 = τ:<LIB:ε,id!>                  v7 = BOR:agree!:<LIB:ε,agree!>
v4 = τ:<LIB:borrowed!,unavailable?>
```

The corresponding adaptor is computed, the new system assessed successfully, and we end up with a system made up of components `LIB`, `BOR`, and their respective adaptors. To check how the approach integrates in a complete development process, the system has been implemented in COM/DCOM using the adaptor models to obtain their code.

5 Related Work

Since Yellin and Strom's seminal paper [27], adaptation techniques [21, 14, 8] have been proposed to correct component mismatch building adaptors. In [11] we made significant advances with an approach supporting name mismatch, system-wide adaptation (more than two components) and event reordering. Yet, all these approaches require the computation of a global adaptor, which is costly, and none supports open systems, which prevents application to pervasive systems.

In [22], component wrappers are composed to augment connector behaviour. This has been revisited in [23], providing automation, but still with a centralised global adaptor as starting point, as for [4] where adaptor distribution is addressed. Several theoretical works have focused on the incremental construction of systems and dynamic reconfiguration [5, 3, 26]. However, these proposals only address syntactic adaptation (via name translation or morphisms) and cannot be used to solve behavioural mismatch. In [19], we have proposed a methodology

to help designers in the incremental construction of component-based systems where adaptors are required. The definition of open systems, their composition and related adaptation algorithms were not supported. Incrementality was achieved using implicit vectors exporting in adaptors all the component services in order to make the design process compatible with adaptation as defined in [11]. This limits the application of [19] at design-time where the set of components to be integrated is known.

6 Conclusion

The integration of software components often requires a certain degree of adaptation. Adaptation approaches have addressed closed systems and the distribution of global adaptors but to our knowledge, none supports open systems. Thus, they are not well suited to systems where components or services may enter and leave at any time, such as pervasive ones. To address this issue, we have proposed here (i) a formalising of component-based open systems which thereafter supports, (ii) an extension of software adaptation to open systems, and (iii) an incremental integration process which avoids the computation of global adaptors. The adaptation solutions we propose are supported by a tool, Adaptor. In its current version, Adaptor can deal with both closed systems and open systems. This tool and its set of validation examples (approx. 70 examples, 25,000 lines of XML specifications) are freely available from [1].

Our main perspectives concern the application of our model-based adaptation techniques to service oriented architectures for pervasive computing. First, relations between adaptation models and implementation languages have to be studied. We have done some experiments using COM/DCOM but Web services are more relevant in this area. The combination of adaptation with semantic composition solutions such as [6] is also an interesting perspective to support not only behavioural but also semantic correctness. To end, end-user composition is a crucial issue in pervasive computing. The support in Adaptor for the use of other adaptation contract formalisms, as presented in Section 2.3 is therefore an interesting perspective.

Acknowledgements. This work has been supported by the French National Network for Telecommunication Research. Adaptor has been developed with S. Beauche. We thank M. Tivoli for the COM/DCOM encoding of the case study and C. Canal for fruitful discussions.

References

1. The Adaptor tool (LGPL licence). Available from P. Poizat's Webpage.
2. G. Agha. Special Issue on Adaptive Middleware. *CACM*, 45(6):30–64, 2002.
3. N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, *LNCS* 2621. Springer.

4. 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*, *LNCS* 4344. Springer.
5. R. J. Back. Incremental Software Construction with Refinement Diagrams. Technical Report 660, Turku Center for Computer Science, 2005.
6. S. Ben Mokhtar, N. Georgantas, and V. Issarny. Ad Hoc Composition of User Tasks in Pervasive Computing Environments. In *Proc. of SC'05*, volume 3628 of *LNCS*. Springer.
7. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42(14), 2004.
8. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
9. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, , and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.
10. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet.*, 12(1):9–31, 2006. Special Issue on Software Adaptation.
11. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, *LNCS* 4037. Springer.
12. H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002.
13. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
14. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
16. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
17. Objet Management Group. Unified Modeling Language: Superstructure. version 2.0, formal/05-07-04, August 2005.
18. P. Poizat. Eclipse Transition Systems. RNRT project STACS deliverable, 2006.
19. P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. In *Proc. of FACS'06*.
20. M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 6(8):10–17, 2001.
21. H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS'02*. Kluwer.
22. B. Spitznagel and D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proc. of ICSE'03*. ACM Press.
23. M. Tivoli and M. Autili. SYNTHESIS, a Tool for Synthesizing Correct and Protocol-Enhanced Adaptors. *L'Objet.*, 12(1):77–103, 2006.
24. S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioural Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.
25. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
26. M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC/FSE'01*. ACM Press.
27. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.