Adaptation of Component Behaviour using Synchronous Vectors

Technical Report ITI-05-10
University of Málaga *

Carlos Canal[1], Pascal Poizat[2], and Gwen Salaün[3]

[1] Department of Computer Science, University of Málaga (Spain)
e-mail: canal@lcc.uma.es
[2] LaMI UMR 8042 CNRS – University of Évry Val d'Essonne (France)
e-mail. poizat@lami.univ-evry.fr
[3] VASY project, INRIA Rhône-Alpes (France)
e-mail: salaun@inrialpes.fr

December 21, 2005

Abstract

Software Adaptation is a crucial issue for the development of a real market of components promoting software reuse. Recent work in this field has addressed several problems related to signature and behavioural mismatch. In this paper, we present our proposal for software adaptation. It builds on previous work overcoming some of its limitations, and makes a significant advance to solve pending issues, especially with the use of regular expressions for governing adaptation rules. Our proposal is based on the use of synchronous vectors, and is supported by dedicated algorithms and tools.

*This work has been partly funded by the European Network of Excellence on AOSD, AOSD-Europe IST-2-004349-NOE.

1 Introduction

Component-Based Software Engineering (CBSE) focuses on composition and reuse, aiming to develop a market of software components, in which customers select the most appropriate software piece depending on its technical specification [BW98]. The development of such a market has always been one of the major concerns of Software Engineering, but it has never become a reality. The reason is that we cannot expect that any given software component perfectly matches the needs of a system where it is trying to be reused. Software is never reused "as it is", and a certain degree of adaptation is always required [NM95].

To deal with these problems a new discipline, *Software Adaptation*, is emerging. Software Adaptation is concerned with providing techniques to arrange already developed pieces of software, in order to reuse them in new systems [CMP04]. Software Adaptation promotes the use of *adaptors* —specific computational entities guaranteeing that components will interact in the right way. Recently, *Software Adaptation* has achieved the status of a definite working area in the field of Software Engineering, and there has been a significant number of research works addressing adaptation issues, and many forums are including it among their topics of interest, if not specifically devoted to adaptation.

CBSE postulates that a component must be reusable from its interface [Szy98], which in fact constitutes its full technical specification. Hence, one of the most important issues to be addressed is how to provide components with a specification that helps in the process of adapting and reusing them. The characteristics and expressiveness of the language used for interface description determines the degree of interoperability we can achieve using it, and the kind of problems that can be solved.

In their book on component-based development [WHS01], Wallnau et al. state that there is a growing gap between the theory and the practice of software design. The theory largely assumes that the design task is to develop specifications for software components; in reality, however, most component-based development relies on preexisting components, which have preexisting specifications, and must be adapted before reuse. With more and more software being developed from commercially available components, it is increasingly critical to recognize the challenges and constraints inherent in such specifications, and to use specific and proven techniques for building component-based systems in a real working environment.

We can distinguish between several levels of interoperability, and accordingly of interface description [CMP06]:

- **Signature level.** Commercial component platforms, such as CCM, J2EE, or .Net, provide convenient ways to describe typed signatures via Interface Description Languages (IDLs) as a means for allowing software composition, even at runtime. Interface descriptions at this level IDLs specify the methods or services that an entity offers (*e.g.*, CORBA-IDL), and sometimes also its external dependencies like in CCM-IDL for OMG's component model [OMG02], or (*e.g.*, WSDL [W3C01] for Web Services).

Typically, these interfaces specify the name of the service, the type of its arguments and return values, and the possible exceptions raised, that is, the full signature of the component. Hence, the kind of problem that we can address at this level is to overcome mismatch in the signature of the components being connected, like mismatch in the names of the services offered or required by the components, or in the number, ordering, or type of their parameters, or even the mismatch caused by the use of heterogeneous programming languages. for developing each component. However, current IDLs are unable to give a real solution to many other interoperability problems. Indeed, even if all signature problems are overcome, there is no guarantee that the components will interoperate suitably, since mismatch may also occur at several different levels not addressed by signature interfaces [BBC05] as provided by current IDLs.

- **Behavioural level.** Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also that it expects from its environment. Behavioural descriptions are required for entities with *state*, that provide non-uniform services, that is, which are not always available, but depend on the internal state of the entity.

  There are several proposals for extending component interfaces with behaviour, thus resulting in what we may call a Behavioural IDL (BIDL) (*e.g.*, WSBPEL [A$^+$05], for describing Web Service choreography).

  A source of mismatch at this level is for instance the lack of exact correspondences between services names. Indeed, there may not be one-to-one correspondences between services offered and required, and one of the components may consider as a single service what will be achieved invoking several services in the other one. Furthermore, the protocols followed by the components may be incompatible (*i.e.*, the components may deadlock when combined), then requiring both protocol transformation and remembering of messages and parameters.

- **Semantic level.** This level describes what the component actually *does* , not only the services it offers, or the messages it exchanges. In fact, even if two components are compatible both at the signature and behavioural levels, present perfectly matching signature interfaces, and they also follow compatible protocols, we cannot ensure that what one of them does when receiving a certain message is what the other one expects. Hence, some kind of functional specification should be provided, which would be particularly interesting for component mining. In the field of Web Services, this level of description is related to the Semantic Web, using OWL-S [OWL04] and other XML-based notations, as an alternative to behavioural descriptions for raising the level of expressiveness of interfaces.

  Nevertheless, the possibilities of adaptation, especially automatic, are less evident at this level (consider for instance transforming a queue into a stack). Another issue would arise if there is no single component performing the required functionality, but two (or more) partially offering it. In

this case, adaptation would involve composing suitably the components found.

- **Service level.** Finally, even if we are able to find a perfect match between components at the signature, behavioural and semantic levels, there is still a number of sources of mismatch, related with non-functional properties like temporal requirements, security, reliability, accuracy, cost, etc. that could make composition impossible. This level of interoperability is the target of Quality of Service (QoS) proposals and their related notations, such as the QoS Modeling Language (QLM) [FK98]. These notations are usually highly customizable, and the possible specifications include mean values, standard deviations and a set of quantiles characterizing the distribution of any self-defined quality metric.

  At this service level, we may have to adapt the component found (provided it offers the right signature, behaviour and functionality) in order to achieve the required QoS. For instance, we may have to use several components in parallel in order to achieve lower response times, or a greater accuracy, or fault tolerance.

The rest of this paper is organized as follows. In Section 2 we survey the more advanced proposals for software adaptation. Then, in Section 3 we formally present our component interface model, and define interface mismatch by means of synchronous products. Section 4 presents our approach to component adaptation, which combines the points in favour of different adaptation approaches, while trying to overcome their limitations. Our proposals for signature and behavioural adaptation are supported by dedicated algorithms, and in both cases the adaptation mappings rely on synchronous vectors. Next, Section 5 extends mappings to take into account regular expressions, enabling complex policies for applying the adaptation vectors. Finally, Section 6 draws up the main conclusions of this work and sketches some future tasks that will be accomplished to extend its results.

2 Related Work

For a thorough review of the state of the art in Software Adaptation, we refer to [CMP06]. Here, we will mention only a few works, those more closely related to our proposal.

One of the first papers explicitly addressing the problems of integrating software components is [GAO95], where architectural mismatch (considered in a very general and broad sense) is presented as caused by the different assumptions that system components make about their own environment. These assumptions are almost always implicit, and quite often they conflict, making them extremely difficult to analyze before building the system. The paper states the need of specific techniques for the analysis of mismatch, and architectural adaptation, and suggests some possible lines of research in this field.

As shown in the previous section, the need for adaptation may (and it probably will) occur at any of the levels of interoperability described, while currently available component platforms address software adaptation only at the signature level. Although some practical issues related with interoperability between different platforms still remain, these are not problems demanding a significant research effort. Hence, most of the recent proposals for adaptation of software have jumped from the signature level to the specification and analysis of behavioural interfaces, promoting the use of BIDLs for describing component protocols.

The foundation for behavioural adaptation was set by Yellin and Strom. In their seminal paper [YS97], they employed finite state machines for specifying component behaviour, and introduced formally the notion of *adaptor* as a software entity capable of enabling the interoperation of two components with mismatching behaviour. They used finite state grammars to specify component interactive behaviour, to define a relation of compatibility, and to address the task of (semi-)automatic adaptor generation.

Some significant limitations of their approach derive from the expressiveness of the notation used, such as the impossibility of representing internal choices or parallel composition of behaviour. Moreover, the asymmetric meaning they gave to input and output actions made it necessary the use of *ex-machina* arbitrators to control system evolutions. Last, but not least, adaptor specifications in [YS97] allowed only to express one-to-one relations between actions, a severe expressiveness bound when facing non-trivial protocol adaptations.

As pointed out in [YS97], the first step needed to overcome behavioural mismatch is to let behaviour information be explicitly represented in component interfaces. In this sense, Process Algebras (PA) feature a very expressive description of interaction protocols, and enable sophisticated analysis of concurrent systems. For these reasons, the use of PA for the specification of component interfaces and for the analysis of component compatibility has been widely advocated.

Taking [YS97] as a starting point, the work of Brogi and collaborators (BBCP) [BBC05, BCP06] presents a methodology for behavioural adaptation. In their proposal, component behaviour is specified using a process algebra — in particular a subset of the π-calculus—, where service offering/invocation is represented by input/output actions in the calculus, respectively. The starting point of the adaptation process is a mapping that states correspondences between services in the interfaces of the components being adapted. This mapping can be considered as an abstract specification of the required adaptor. Then, an adaptor generation algorithm refines the specification given by the mapping into a concrete adaptor implementation, taking also into account the behavioural interfaces of the components, which ensures correct interaction between them according to the mapping. The adaptor is able to accommodate not only syntactical mismatch between service names, but also the interaction protocols that the components follow (*i.e.*, the partial ordering in which services are offered/invoked).

Another interesting proposal in this field is that of Inverardi and Tivoli

(IT) [IT03a]. Their proposal goes beyond BBCP specifying and analyzing a set of properties, addressing how to enforce certain behavioural properties (namely deadlock-freedom) out of a set of already implemented behaviors. The software architecture imposed on the assembly allows for detection and recovery of component integration anomalies. Starting from the specification with MSCs of the components to be assembled and of the properties that the resulting system should verify (liveness and safety properties expressed as specific processes), IT develop a framework which automatically derive the adaptor glue code for the set of components in order to obtain a property-satisfying system.

The IT proposal has been extended in [IT03b] with the use of temporal logic; coordination policies are expressed as LTL properties, and then translated into Büchi automata. Note that, as deadlock freedom is expressible in temporal logics, works dealing with the second approach could be applied for deadlock freedom, too. Another important thing to notice is that this approach is fully tool equipped.

A comparison between both proposals yields the following results:

- **Description of behaviour.** The BBCP approach is based on the use of process algebra, while IT takes MSCs as the starting point for obtaining automata descriptions. Both automata-like languages and process algebras have well-known formal operations (equivalences, refinements, model-checking) implemented in formal tools (for example, NuSMV, SPIN, LTSA, CADP). Automata are nice models for developing formal algorithms (for equivalence, model checking, subtyping, etc.), and are also more user-friendly. Process algebras are more abstract, concise and expressive, but at the same time more complex to use and verify, which is the reason why most verification tools for process algebras first proceed by a translation into labelled transition systems (LTS).

- **Adaptor specification.** The Büchi automata in the IT proposal may be considered as a sort of adaptor specification, but the use of MSCs requires an exact correspondence between action names in the components being adapted. Hence, the notion of mapping is not present in IT, which does not address a number of significant adaptation issues, like mismatching service names, description of data types, in events (allowing then for parameter reconfiguration), or one-to-many correspondences between services. On the contrary, all these are covered in the BBCP proposal, where the use of a mapping not only enables to define one-to-one correspondences between single service names, but also more complex correspondences between whole sequences of services.

- **Protocol adaptation.** IT adaptors are only able to enforce behaviour by adopting a *restrictive* approach. The idea is to restrict the interactive behaviour of the components by cutting off undesired branches (*e.g.*, those leading to deadlock) in the derivation tree generated by composing component protocols. Hence, mismatching behaviour is not accommodated, but just avoided. The technique consists on synthesizing a controller between

the components (using expectations of these components with reference to their environment). Then, the controller is used to restrict the behaviour of the components. Deadlocks detected on the controller are avoided by removing all its finite branches. In this way, the controller enables the maximum set of interactions between the components which do not lead to deadlock.

Moreover, IT does not address behavioural adaptation properly, since the transmission of information between components is performed in a synchronous way, without being able to memorize or perform reordering of messages and parameters as in BBCP. In fact, the BBCP approach can be considered as *generative*, since it eventually enables to recombine the behaviour of the components by means of the adaptor. This is a more general approach, since it takes into account the fact that when two components have been developed separately, they often disagree not only on the names of their services, but also in the protocol for accessing these services.

Notice that in both cases, these techniques are not invasive, and therefore they are suitable for black-box components.

- **Property satisfaction.** The only properties ensured by the BBCP approach are *(i)* deadlock-freedom, ensuring that after adaptation component interfaces are consistent, and *(ii)* the satisfaction of the correspondences between actions established in the mapping. On the contrary, the IT proposal has in favour the use of LTL for specifying several kinds of properties other than deadlock. These properties will be enforced during generation, constraining the resulting adaptor. In fact, we may say that BBCP mappings are *underspecified* in the sense that they do not determine a particular adaptor, but only some very general rules that any correct adaptor must satisfy. As a consequence, the generation algorithm may return an ill-constructed adaptor, that although ensures deadlock-freedom and satisfies action correspondences in the mapping, is not suitable for real use.

- **Binary vs. system-wide adaptation.** The BBCP proposal only addresses binary adaptors (allowing the interaction of just two mismatching parts), while the IT proposal is system-wide, dealing with adaptation between any number of components.

3 Interfaces and Mismatch

3.1 Component Interfaces

Component interfaces are given using a signature and a behavioural interface.

Definition 1 (Signature) *A* signature Σ *is a set of operation profiles. This set is a disjoint union of* provided *operations and* required *operations. An*

operation profile is simply the name of an operation, together with its argument types, its return type and the exceptions it raises.

This definition naturally corresponds to the signature definitions in component based models such as CCM or J2EE. Such signatures are defined using an IDL. In this paper we do not deal with operation arguments, return values or exceptions.

We also take into account behavioural interfaces through the use of labelled transition systems (LTS).

Definition 2 (LTS) *A* Labelled Transition System *is a tuple* (A, S, I, F, T) *where:* A *is an alphabet (set of events),* S *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are final states, and* $T \subseteq S \times A \times S$ *is the transition function.*

The alphabet of the LTS is built on the 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 $r!$. As in CCS, $(a, \bar{a})$ denote complementary actions —*i.e.*, if a is $p?$ (respectively $r!$), then $\bar{a}$ is $p!$ (respectively $r?$). Additionally, the ε event is used in LTS to denote a do-nothing event. This is a model-level event that must not be confused with the τ action in process algebras, which denotes an internal evolution.

LTSs are adequate as far as user-friendliness and development of formal algorithms are concerned. However, higher-level behavioural languages such as process algebras can be used to define behavioural interfaces in a more concise way. Taking advantage of the fact that process algebraic descriptions can be translated into LTS models, we use as a BIDL the part of the CCS notation restricted to sequential processes: `P ::= 0 | a?.P | a!.P | P1+P2 | A`, where `0` denotes a do-nothing process, `a?.P` a process which receives `a` and then behaves as `P`, `a!.P` a process which sends `a` and then behaves as `P`, `P1+P2` a process which may act either as `P1` or `P2`, and `A` denotes the call to a process defined by an agent definition equation `A = P`.

As process algebras do not enable to define initial and final states, we extend this CCS notation to tag processes with initial (`i`) and final (`f`) attributes. Finally, `0` is often omitted in processes (*e.g.*, `a!.b![f]` is used for `a!.b!.0[f]`).

Example 1 *Consider a client that repetitively sends a query and its argument, and then waits for an acknowledgement, quitting with an* `end!`*, and a server repetitively waiting for a query and a value, then returning a given service:*

```
Client[i] = query!.arg!.ack?.Client + end![f]
Server[i,f] = query?.value?.service!.Server
```

The LTSs for these two components are given below with initial and final states marked using the UML state diagrams notation.

3.2 Behavioural Mismatch

Various definition of behavioural mismatch have been proposed in the field of software adaptation and software architecture analysis [CMP06]. We build on

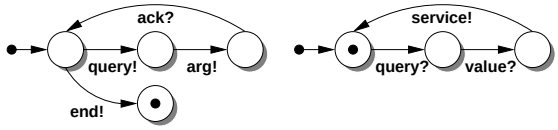

Figure 1: A simple Client/Server system

the most commonly accepted one, namely deadlock-freedom. The first step is to define the semantics of a system made up of several identified components. This semantics can be given, following work by Arnold [Arn94] using synchronous product.

Definition 3 (Synchronous Product) *The* synchronous product *of n LTS $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in 1..n$, is the LTS (A, S, I, F, T) such that:*

- $A \subseteq \Pi_{i \in 1..n} A_i$, $S \subseteq \Pi_{i \in 1..n} S_i$, $I = (I_1, \ldots, I_n)$,
- $F \subseteq \{(s_1, \ldots, s_n) \in S \mid \bigwedge_{i \in 1..n} s_i \in F_i\}$,
- *T is defined using the following rule:*
  $\forall (s_1, \ldots, s_n) \in S$, $\forall i, j \in 1..n, i < j$ *such that*
  $\exists (s_i, a, s'_i) \in T_i$, $\exists (s_j, \bar{a}, s'_j) \in T_j$, *then*
  $(x_1, \ldots, x_n) \in S$ *and* $((s_1, \ldots, s_n), (l_1, \ldots, l_n), (x_1, \ldots, x_n)) \in T$, *where*
  $\forall k \in 1..n$, $l_k = \{\ a$ if $k = i$, $\bar{a}$ if $k = j$, ε *otherwise* $\}$
  $x_k = \{\ s'_i$ if $k = i$, s'_j if $k = j$, s_k *otherwise* $\}$

We are now able to characterize behavioural mismatch by means of deadlock.

Definition 4 (Deadlock State) *Let $L = (A, S, I, F, T)$ be a LTS. A state s is a deadlock state for L, noted* `dead`(s)*, iff it is in S, not in F and has no outgoing transitions: $s \in S \wedge s \notin F \wedge \nexists l \in A, s' \in S \,.\, (s, l, s') \in T$.*

Definition 5 (Deadlock Mismatch) *An LTS $L = (A, S, I, F, T)$ presents a deadlock mismatch if there is a state s in S such that* `dead`(s).

To check if a system made up of several components presents behavioural mismatch, its synchronous product is computed and then Definition 5 is used.

Example 2 *Taking Example 1, we obtain the following synchronous product:*

Figure 2: Synchronous product for the Client/Server system in Figure 1

Note that the deadlock is caused by (i) the client required service `end!` which has no counterpart in the server, and (ii) name mismatching between the client required service `arg!` and the server provided service `value?`.

We may now define what is a correct adaptor for a system. An adaptor is given by an LTS which, put into a non-deadlock-free system yields a deadlock-free one. For this to work, the adaptor has to preempt all the component communications. Therefore, prior to the adaptation process, component service names may have to be renamed prefixing them by the component name, *e.g.*, `c:service!`.

The product we have defined here is common in the community and hence is supported by tools such as the CADP toolbox [GLM02]. Our deadlock definition however is slightly different from the one used in these tools, since it has to distinguish between success (deadlock in a final state), and failure (deadlock in a non-final state). Mismatch detection can be automatically checked by CADP up to the adding within component interfaces of specific loop transitions labelled with `accept` over final states. Then the EXP.OPEN tool [Lan05] of CADP is used to perform a full matching product between the component interfaces:

1. we first translate each component interface C_i into a LTS in the `.aut` format in a file `C_i.aut`,
2. we make a synchronizing file `Global.exp` with pattern:
   ```
   C_1 || C_2 || ...  || C_n
   ```
3. we use EXP.OPEN to compute the resulting global LTS (in `.aut` format).

The same process can be used to check the correctness of adapted systems. Components are prevented to synchronize between them whereas the adapter is fully synchronized with them. With reference to the enumeration above, this means that:

1. the adapter is also translated into a `.aut` file, `A.aut`
2. the `Global.exp` pattern is now:
   ```
   (C_1 ||| C_2 ||| ...  ||| C_n) || A
   ```

In both cases, the presence of a deadlock state in the products means a behavioural mistmatch. States with accept loops over them are correct final states for the global system.

In Figure 3 we give an overview of our proposal for adaptation. Component interfaces are used to check for behavioural mismatch. If there is a mismatch then adaptation has to be performed. First, signature adaptation can be used if no reordering of services is needed. This is described in Section 4.2. If this fails, then behavioural adaptation, taking into account reordering can be performed as described in Section 4.3. In both cases, our procedures rely on an abstract description of the adaptor properties using vectors. These procedures will be then extended in Section 5 to take into account regular expressions over adaptation vectors, enabling to take into account more expressive adaptor properties.

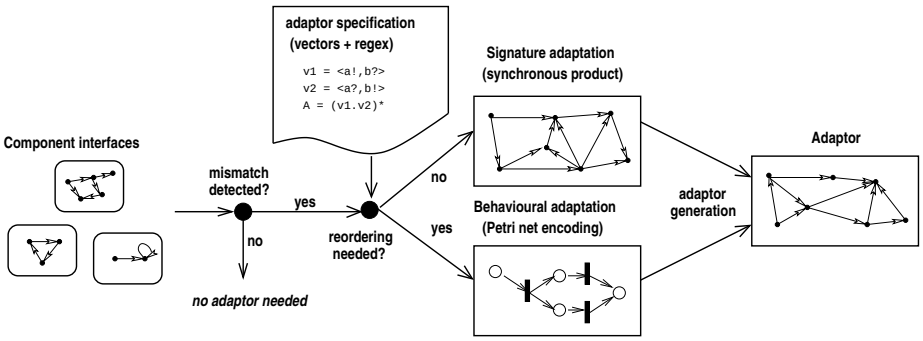

Figure 3: Our adaptation proposal.

4 Adaptation based on Synchronous Vectors

4.1 Synchronizing with Vectors

The first thing to solve in adaptation is signature mismatch. Our idea is to use synchronous vectors as a way to denote a morphism between event names in different components.

Vectors generalize synchronous product by expressing not only synchronization between processes on the same event names (a and $\bar{a}$ in Definition 3), but more general correspondences between the events of the process involved.

Definition 6 (Vector) *A* synchronous vector *(or* vector *for short) for a set of Id indexed components $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in Id$, is a tuple (e_i) with $e_i \in A_i \cup \{\varepsilon\}$.*

Note that vectors are simple correspondences between events. Extensions can be easily defined to consider relations between events with data.

Definition 7 (Synchronous Vector Product) *The* synchronous vector product *of n LTS $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in 1..n$ with a set of vectors V, is the LTS (A, S, I, F, T), denoted by $\Pi(L_i, V)$, such that:*

- $A \subseteq \Pi_{i \in 1..n} A_i$, $S \subseteq \Pi_{i \in 1..n} S_i$, $I = (I_1, \ldots, I_n)$,
- $F \subseteq \{(s_1, \ldots, s_n) \in S \mid \bigwedge_{i \in 1..n} s_i \in F_i\}$,
- *T is defined using the following rule:*
  $((s_1, \ldots, s_n), (l_1, \ldots, l_n), (s'_1, \ldots, s'_n)) \in T$ and $(s'_1, \ldots, s'_n) \in S$ if
  $\exists (s_1, \ldots, s_n) \in S$ and $\exists v = (l_1, \ldots, l_n) \in V$ such that,
  $\forall l_i \in v$ $s'_i = s_i$ if $l_i = \varepsilon$ and $\exists (s_i, l_i, s'_i) \in T_i$ otherwise.

4.2 Signature Adaptation

We first address adaptation where only signature mismatch is taken into account. Our algorithm takes as input the Id indexed set of components LTS L_i of the systems and a mapping which is a synchronous vector V.

1. compute the product $P = (A_P, S_P, I_P, F_P, T_P) = \Pi(L_i, V)$
2. obtain $P_{\text{restr}} = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_{P_{\text{restr}}})$ from P recursively removing transitions and states yielding deadlocks: find a state s such that $\texttt{dead}(s)$, remove s and any transition t with target s, and do this until there is no more such s in the LTS.
3. from P_{restr}, build the adaptor $A = (A_{P_{\text{restr}}}, S_{P_{\text{restr}}} \cup S_{\text{add}}, I_{P_{\text{restr}}}, F_{P_{\text{restr}}}, T_A)$ where S_{add} and T_A are defined as follows.

   For each $t = (s = (s_1, \ldots, s_n), (l_1, \ldots, l_n), s' = (s'_1, \ldots, s'_n))$ in $T_{P_{\text{restr}}}$, let $L_{\text{rec}} = \{l? \mid l! \in (l_1, \ldots, l_n)\}$ and $L_{\text{em}} = \{l! \mid l? \in (l_1, \ldots, l_n)\}$. Let then Seq_{rec} be the set of all permutations over L_{rec} and Seq_{em} be the set of all permutations over L_{em}. For each couple (R, E) in $\text{Seq}_{\text{rec}} \times \text{Seq}_{\text{em}}$, $R = (r_1, \ldots, r_{nr})$ and $E = (e_1, \ldots, e_{ne})$, seq $= (r_1, \ldots, r_{nr}, e_1, \ldots, e_{ne})$, construct the transaction

   $$s = q_0 \overset{\text{seq}[1]}{\rightarrow} q_1 \ldots q_k \overset{\text{seq}[k+1]}{\rightarrow} q_{k+1} \ldots q_{n-1} \overset{\text{seq}[n]}{\rightarrow} s' = q_n$$

   adding each $q_{k \in 1..n-1}$ in S_{add} and each $q_k \overset{\text{seq}[k+1]}{\rightarrow} q_{k+1}$ $(k \in 0..n)$ in T_A.

This algorithm builds the most general adaptor in the sense that it simulates any other adaptor for the mismatching system.

4.3 Behavioural Adaptation

Let us now extend the domain of adaptation problems we deal with. The goal is to address not only signature, but also behavioural mismatch, that is, the incompatible ordering of the events exchanged. Indeed, our signature adaptation proposal above (and also the approach presented in IT works) would yield an empty adaptor in presence of behavioural mismatch, concluding that adaptation is not possible. In this case, the adaptation process may try to reorder protocol events in-between the components. To this purpose, we present a second approach which complements the one above. However, it does not replace it as the process may not agree on message reordering.

Our behavioural adaptation approach is based on previous works dedicated to the analysis of component queue boundedness [MPR04]. In order to accomodate behavioural mismatch, the events received by the adaptor are desynchronized from their emission. Our algorithm can be simulated by a translation of the problem into Petri nets [Mur89]. The main advantage of such an approach is that it is equipped with efficient tools.

We first proceed by constructing a Petri net representation of the assumptions the components make on their environment (by mirroring their behavioural

interfaces), and then build causal dependences between the events received and sent by the adaptor accordingly to the mapping, given under the form of synchronous vectors. This allows us to build an adaptor which accommodates both signature and behavioural mismatch.

1. for each component i with LTS L_i, for each $s_j \in S_i$, add a place `Control-i-s_j`
2. for each component i with initial state I_i, put a token in `Control-i-I_i`
3. for each $a!$ in $\bigcup_i A_i$, add a place `Rec-a`
4. for each $a?$ in $\bigcup_i A_i$, add a place `Em-a`
5. for each component i with LTS L_i, for each $(s, l, s') \in T_i$:
   - add a transition with label $\bar{l}$, one arc from place `Control-i-s` to the transition and one arc from the transition to place `Control-i-s'`
   - if `l` has the form $a!$ then add one arc from the transition to place `Rec-a`
   - if `l` has the form $a?$ then add one arc from place `Em-a` to the transition
6. for each vector $v = (l_1, \ldots, l_n)$ in V:
   - add a transition with label `tau`
   - for each l_i with form $a!$, add one arc from place `Rec-a` to the transition
   - for each l_i with form $a?$, add one arc from the transition to place `Em-a`
7. for each tuple $(f_1, \ldots, f_n)$, $f_i \in F_i$, of final states, add a (loop) `accept` transition with arcs from and to each of the tuple f_i

Once this Petri net encoding has been performed, we compute its marking graph. If it is finite (*e.g.*, for non recursive adaptors) then it gives a behavioural description of the adaptor. If not (it cannot be computed in finite time), then we compute the cover graph of the net. Note that due to the overapproximation of such a graph, we add a guard `[#Em-a>1]` (`#Em-a` meaning the number of tokens in place `Em-a`) on any `a!` transition in this graph leaving a state where `#Em-a` is ω. In both cases (marking or cover graph), step 2 of the algorithm in Section 4.2 has to be performed on the adaptor obtained.

This algorithm is supported by tools. We have made successful experiments with the TINA tool [BRV04] to generate marking and cover graphs. Our approach yields graphs which can be too large for a human reader. We simplify the adaptor LTS passing the resulting output file to CADP and performing a $\tau * a$ reduction on it to remove the meaningless `tau` transitions it contains.

4.4 Application

We first demonstrate algorithm 4.2 on an example where only signature adaptation is to be taken into account. For this purpose, we reuse example 1 whose LTS are given in Figure 1.

Example 3 *Taking into account renaming, we have:*

`Client[i] = c:query!.c:arg!.c:ack?.Client + c:end![f]`

`Server[i,f] = s:query?.s:value?.s:service!.Server.`

The vectors in this case are <`c:query!,s:query?`>, <`c:arg!,s:value?`>, <`c:ack?,s:service!`> *and* <`c:end!,s:`ε>. *Step 1 of the algorithm build the product in Figure 4. No deadlocks are to be removed by step 2 in this case. Then, step 3 builds the adapter given in Figure 5.*

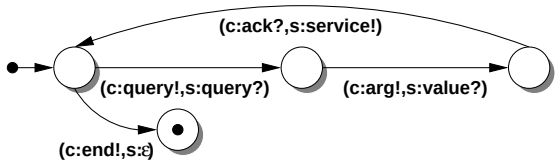

Figure 4: Synchronous vector product for example 3

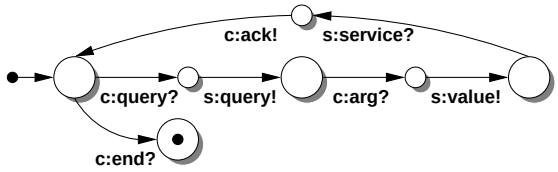

Figure 5: Adapter for example 3

We now present an example following the behavioural adaptation technique above.

Example 4 *Suppose we have a client* `Client[i]=req!.arg!.ack?[f]` *and a server* `Server[i]=value?.query?.service![f]` *with vectors* <`req!,query?`>, <`arg!,value?`> *and* <`ack?,service!`>. *Such an example is typical of clients and servers which follow different standards for the order of sending subservice elements. The Petri net encoding (see Section 4.3) of the system is:*

Computing the marking graph, we obtain a LTS with 13 states and 16 transitions (Fig. 7, left), which once reduced yields the correct adaptor (Fig. 7, right)[1].

We want to stress that our adaptation proposal is an *automatic* process. For the sake of the presentation, we have shown here simple examples for which

[1]Note the `i` which stands in CADP for `tau` transitions, and the `accept` loop transitions which enable the detection of correct final states. Figures are generated by CADP.

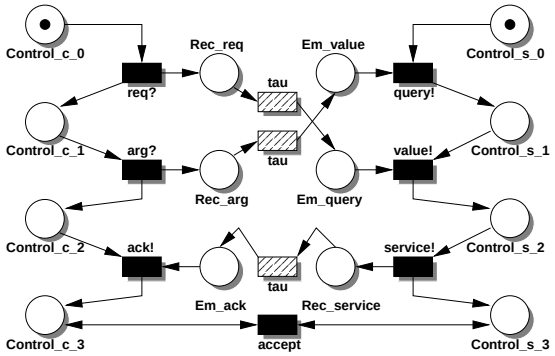

Figure 6: Petri net encoding of a simple Client/Server system.

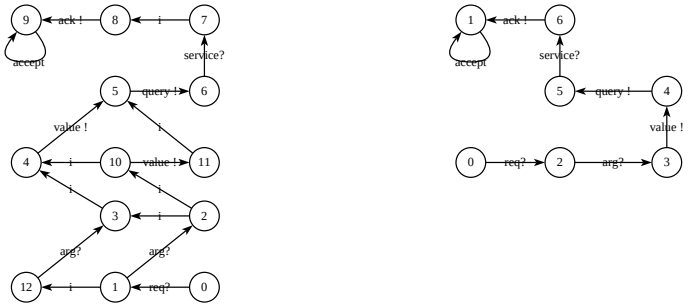

Figure 7: Initial and reduced adaptor for the Client/Server system.

the adaptor could be obtained manually. However, using slightly more complex component protocols, the adaptor becomes too large to be obtained by hand. Moreover, the use of regular expressions in the next section will increase the complexity of the adapting process and the need for such automatic techniques.

5 Adaptation Patterns

In this section, we tackle the problem of adaptation mappings which may change over time. In the following, we present a way to express such mappings using regular expressions (regex), and then update our algorithms to deal with them.

5.1 Vector Regular Expressions (Regex)

First, we introduce the syntax for regex. These will be used in place of the basic vector mappings we presented in Section 4.

Definition 8 (Vector Regex) *Given n LTS $L_i = (A_i, S_i, I_i, F_i, T_i)$, and a set of vectors $V = \{(e_{ij})\}_j$ for their adaptation, with $e_{ij} \in A_i \cup \{\varepsilon\}$, a (vector) regex*

for these LTS can be generated by the following syntax: `R ::= v (VECTOR)| R1.R2 (SEQUENCE)| R1+R2 (CHOICE)| R* (ITERATION)`, *where* `R`, `R1`, `R2` *are regex, and* `v` *is a vector in* V

Example 5 (Alternating use client) *Suppose we have a system formed by one client* `C` *and two servers,* `S` *and* `A`*:*

`C[i] = end![f] + req!.arg!.ack?.C`,
`S[i,f] = value?.query?.service!.S`, *and*
`A[i,f] = value?.query?.service!.A`.

One may want to express in the adaptation mapping that the client accesses the two servers alternatively, and not always the same one. For this, we use the following regex: $(v_{s1}.v_{s2}.v_{s3}.v_{a1}.v_{a2}.v_{a3})*.v_{\text{end}}$ *with*

v_{s1} =<`req!,query?,`ε>, v_{a1} =<`req!,`ε`,query?`>, v_{end} =<`end!,`ε`,`ε>,
v_{s2} =<`arg!,value?,`ε>, v_{a2} =<`arg!,`ε`,value?`>,
v_{s3} =<`ack?,service!,`ε>, v_{a3} =<`ack?,`ε`,service!`>.

Example 6 (Connected *vs* **non connected modes)** *Suppose a client/server system where the client* `C` *sends its id only once at login time, while the server* `S` *requires an identification every time the client does a request. Here we have:*

`C[i]=log!.Logged`, *with*
`Logged[f]=req!.ack?.Logged`, *and*
`S[i,f]=log?.req?.ack!.S`

The regex for describing the adaptation required is now $v_0.v_2.v_3.(v_1.v_2.v_3)*$ *with* v_0 =<`log!,log?`>, v_1 =<ε`,log?`>, v_2 =<`req!,req?`> *and* v_3 =<`ack?,ack!`>.

5.2 Signature Adaptation

To be able to update our algorithms for using our new regex mappings[2], we first define how to obtain an LTS from them. This corresponds to the obtaining of an automaton which recognizes the language of a regex. It is a well known problem, see Figure 8 for its principles and [HU79] for a complete description of it. The only difference is that the atoms of our regex are vectors and not elements of basic alphabets.

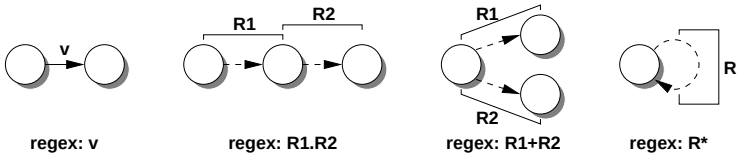

Figure 8: Regex translation principles (intuition)

In Figure 9 we give examples of regex translations for v_0+v_1.(v_2+v_3.v_4)*, Example 5 and Example 6, respectively.

[2]Note that our new algorithms would apply to the vector mappings we have defined in the previous section, just taking the set $V = \{v_i\}$ of vectors as the regex $(v_1 + v_2 + \ldots + v_n)*$.

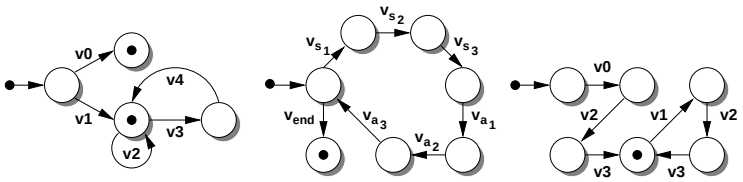

Figure 9: Regex translation examples

Hence, instead of using a regex, one may also use directly the LTS that derives from such regex, (*i.e.*, an LTS where the alphabet corresponds to vectors).

We then modify the synchronous vector product to take a regex LTS in place of the vector argument.

Definition 9 (Synchronous Vector Product (with regex LTS)) *The* synchronous vector product (with regex LTS) *of* n *LTS* $L_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in 1..n$ *with a regex LTS* $L_R = (A_R, S_R, I_R, F_R, T_R)$, *is the LTS* (A, S, I, F, T) *such that:*

- $A \subseteq A_R \times \Pi_{i \in 1..n} A_i$, $S \subseteq S_R \times \Pi_{i \in 1..n} S_i$, $I = (I_R, I_1, \ldots, I_n)$,
- $F \subseteq \{(s_r, s_1, \ldots, s_n) \in S \mid s_r \in F_R \wedge \bigwedge_{i \in 1..n} s_i \in F_i\}$,
- T *is defined using the following rule:*
  $((s_r, s_1, \ldots, s_n), (l_r, l_1, \ldots, l_n), (s'_r, s'_1, \ldots, s'_n)) \in T$ *and* $(s'_r, s'_1, \ldots, s'_n) \in S$ *if*
  $\exists (s_r, s_1, \ldots, s_n) \in S$ *and* $\exists v = (s_r, (l_{r_1}, \ldots, l_{r_n}), s'_r) \in T_R$ *with,*
  $\forall l_{r_i}\ s'_i = s_i$ *if* $l_{r_i} = \varepsilon$ *and* $\exists (s_i, l_{r_i}, s'_i) \in T_i$ *otherwise.*

To apply the Section 4.2 algorithm we just have now to *discard* the first element of the product components, that is, from the LTS $L = (A, S, I, F, T)$ obtain the LTS $L' = \texttt{proj}(L) = (A', S', I', F', T')$ such that $\forall X \in \{A, S, I, F\}\ X' = \{\texttt{cdr}(x) \mid x \in X\}$ and $T' = \{(\texttt{cdr}(s), \texttt{cdr}(l), \texttt{cdr}(s')) \mid (s, l, s') \in T\}$ with $\texttt{cdr}((x_0, x_1, \ldots, x_n)) = (x_1, \ldots, x_n)$.

We may now modify the algorithm for signature mismatching in Section 4.2. The new algorithm takes as input the Id indexed set of components LTS L_i of the system and a mapping which is a regex R (for the set of LTS). We just have to replace step 1 in this algorithm by:

1. compute the LTS L_R for the regex R
2. compute the product $P_R = (A_{P_R}, S_{P_R}, I_{P_R}, F_{P_R}, T_{P_R}) = \Pi(L_R, L_i)$
3. compute $P = \texttt{proj}(P_R)$

5.3 Behavioural Adaptation

Our algorithm for behavioural adaptation can also be adapted to deal with regex.

1. compute the LTS $L_R = (A_R, S_R, I_R, F_R, T_R)$ for the regex R.
2. build the Petri net encoding for the problem as presented in section 4.3, replacing part 6 with:
   - for each state s_R in S_R, add a place `ControlR-s_R`
   - put a token in place `ControlR-I_R`
   - for each transition $t_R = (s_R, (l_1, \ldots, l_n), s'_R)$ in T_R:
     - add a transition with label `tau`, one arc from place `ControlR-s_R` to the transition and one arc from the transition to place `ControlR-s'_R`
     - for each l_i which has the form $a!$, add one arc from place `Rec-a` to the transition
     - for each l_i which has the form $a?$, add one arc from the transition to place `Em-a`
3. in the building of `accept` transitions, add F_R to the F_i taken into account (final states now correspond to acceptance states of the regex LTS).

The rest of the algorithm (computing marking or cover graph, and reducing them) is the same.

5.4 Application

We here develop Examples 5 and 6 above, following our behavioural adaptation technique.

Example 7 (Example 5 developed) *First note that, as explained before, we rename arguments to avoid name clash. We have:*

`C[i] = c:end![f] + c:req!.c:arg!.c:ack?.C`,
`S[i,f] = s:value?.s:query?.s:service!.S`, *and*
`A[i,f] = a:value?.a:query?.a:service!.A`.

To express that the client alternatively uses the two servers we may use the following regex: $R_1 =$ `(`v_{s1}`.`v_{s2}`.`v_{s3}`.`v_{a1}`.`v_{a2}`.`v_{a3}`)*.`v_{end} *with:*

$v_{s1} =$<`c:req!,s:query?,`ε>, $v_{a1} =$<`c:req!,`ε`,a:query?`>,
$v_{s2} =$<`c:arg!,s:value?,`ε>, $v_{a2} =$<`c:arg!,`ε`,a:value?`>,
$v_{s3} =$<`c:ack?,s:service!,`ε>, $v_{a3} =$<`c:ack?,`ε`,a:service!`>,
$v_{end} =$<`c:end!,`ε`,`ε>

Note that this mapping is probably overspecified, since it imposes a strict alternation between servers. Instead, one may choose to authorize the client to access any server it wants. Then, the mapping becomes:

$R_2 =$ `(`v_{s1}`.`v_{s2}`.`v_{s3} `+` v_{a1}`.`v_{a2}`.`v_{a3}`)*.`v_{end}

We have run both examples and obtained (after reduction) the adaptors in Fig. 10 (left for R_1*, and right for* R_2*.) Note that applying step 2 of the algorithm presented in Section 4.2, the state* `1` *and the corresponding transition are removed for* R_1*. Both adapters solve the existing mismatch, making the system deadlock-free, as demonstrated in Figure 11 for* R_2*.*

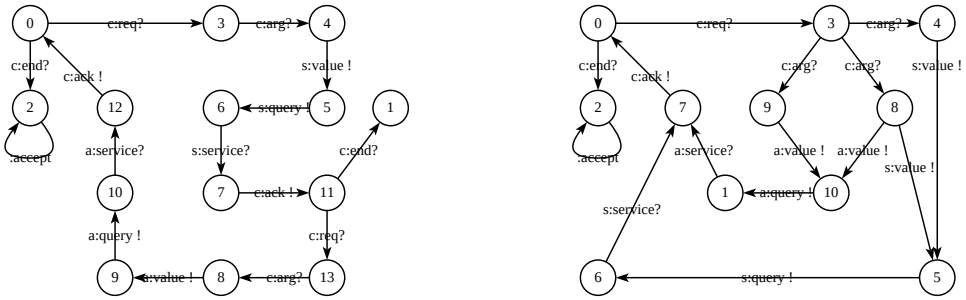

Figure 10: Adaptors obtained for the alternating Client/Server system.

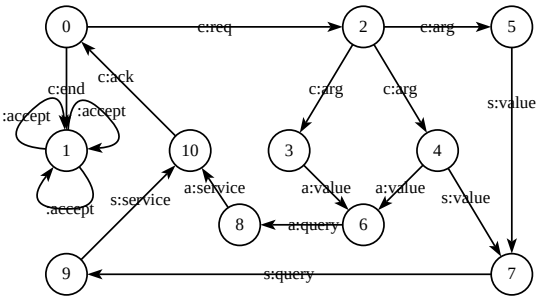

Figure 11: Global system used to check for behavioural mismatch.

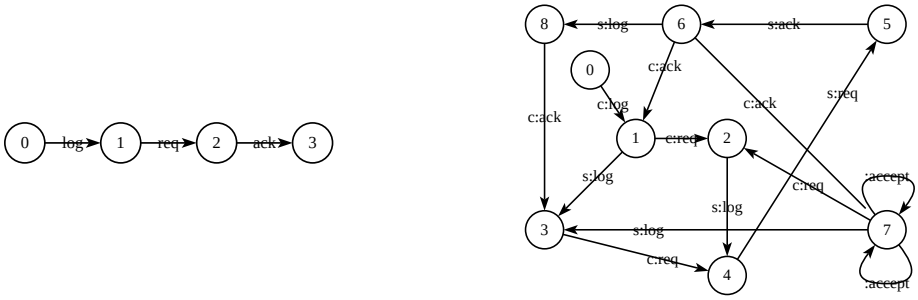

Figure 12: Global systems used to check for behavioural mismatch.

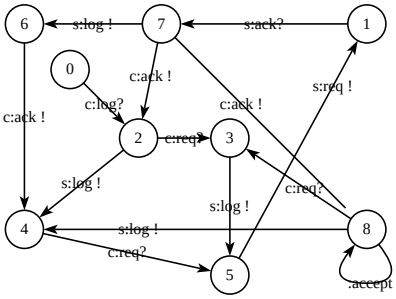

Figure 13: Adaptor obtained for the connection/no connection Client-Server system.

Example 8 (Example 6 developed) *Again, we use renaming.*
`C[i]=c:log!.Logged`*, with*
`Logged[f]=c:req!.c:ack?.Logged`*, and*
`S[i,f]=s:log?.s:req?.s:ack!.S`
The regex for describing the adaptation required is now $v_0.v_2.v_3.(v_1.v_2.v_3)*$ *with* $v_0 =$<`c:log!,s:log?`>, $v_1 =<\varepsilon$,`s:log?`>, $v_2 =$<`c:req!,s:req?`> *and* $v_3 =$<`c:ack?,s:ack!`>. *We first check in Figure 12, left, that the system deadlocks. Running the example and after deadlock removal and reduction, we obtain the adapter in figure 13. We can then check in Figure 12, right, that the adapted system (made up of the client, the server, and the adapter) is free from behavioural mismatch.*

6 Comparison and Conclusions

Software Adaptation has become a crucial issue for the development of a real market of components enhancing software reuse. Recent research work in this

field —in particular that of BBCP and IT [BBC05, BCP06, IT03a, IT03b]— has addressed several problems related to signature and behavioural mismatch.

In this paper, we have shown our proposal for software adaptation. It builds on these previous works, overcoming some of their limitations, and making a significant advance to solve some of the pending issues.

Our approach addresses system-wide adaptation (*i.e,*. differently from BBCP, it may involve more than two components). It is based on LTS descriptions of component behaviour, instead of process algebra as in BBCP. However, we may also describe behaviour by means of a simple process algebra, and use its operational semantics to derivate LTS from it. Differently from IT, we use synchronous vectors for adaptor specification, playing a similar function than the mappings rules in BBCP. With that, we are able to perform signature adaptation.

With respect to behavioural adaptation, our approach can be considered as both restrictive and generative, since we address behavioural adaptation by enabling message reordering (as in BBCP), while we also remove incorrect behaviour (as in IT). Similarly to both approaches, our main goal is to ensure deadlock freedom. However, more complex adaptation policies and properties can be specified by means of regular expressions.

Indeed, the most relevant achievement of our proposal is this use of regular expression for imposing additional properties over mappings. In fact, the semantics of BBCP mappings can be expressed by combining their different rules (in our case, vectors) in a regular expression by means of the choice (+) operator. On the contrary, our regex are much more expressive, solving the problem of BBCP underspecified mappings [BBC05], and allowing to take into account a new class of adaptation problems.

In Table 1 we give a synthesis of the features of our approach compared to IT and BBCP.

criteria	IT	BBCP	our proposal
behaviour description language	automata	proc. algebra	LTS or proc. algebra
properties	no deadlock, LTL properties	no deadlock —	no deadlock regular expressions
mappings/adaptor abstraction	no	yes	yes
name mismatch	no	yes	yes
data types	no	yes	no
message reordering	no	yes	yes
system-wide adaptation	yes	no	yes

Table 1: Comparison of Adaptation approaches

There are still some open issues in our proposal, deserving future work. First, and differently from BBCP, we do not deal with data types, nor with one-to-many correspondences between services. Taking data into account would require more expressive models than LTS, such as Symbolic Transition Systems (STS) [MPR04]. This is a perspective for our work, since it allows the descrip-

tion of the data involved in the operations within the protocol without suffering from the state explosion problem that usually occurs in process algebraic approaches.

With respect to one-to-many correspondences between services (one of the strong points in favour of the BBCP proposal), we intend to explore how regular expressions can be used for that purpose. More expressive models for mappings, such as non-regular protocols [Süd05], could also be extended to vectors in order to get a bigger class of properties expressible at the adaptor level (*e.g.* load-balancing adaptation of the access of clients to servers).

Finally, we intend to implement our adaptation algorithms in ETS, an Eclipse plug-in that we have developed for the experimentation over LTS and STS [Poi06].

References

[A+05] T. Andrews et al. *Business Process Execution Language for Web Services (WSBPEL).* BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, February 2005. Available at `http://www.ibm.com/developerworks/library/specification/ws-bpel`.

[Arn94] A. Arnold. *Finite Transition Systems.* International Series in Computer Science. Prentice-Hall, 1994.

[BBC05] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[BCP06] A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 2006. To appear. A previous version of this work was published as *Soft Component Adaptation*, ENTCS 85(3), Elsevier, 2004.

[BRV04] 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. TINA web page: `http://www.laas.fr/tina/`.

[BW98] A.W. Brown and K.C. Wallnau. The Current State of CBSE. *IEEE Software*, 15(5):37–47, 1998.

[CMP04] C. Canal, J.M. Murillo, and P. Poizat. Coordination and Adaptation Techniques for Software Entities. In *ECOOP 2004 Workshop Reader*, volume 3344 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2004.

[CMP06] C. Canal, J.M. Murillo, and P. Poizat. Software adaptation: an introduction. *L'Objet. Special Is-*

sue on Software Adaptation, 12(1), 2006. To appear. http://www.lami.univ-evry.fr/~poizat/documents/publications/CMP06.pdf.

[FK98] S. Frølund and J. Koistinen. Quality-of-service specification in distributed object systems. Technical Report Technical Report HPL-98-159, Hewlett Packard. Software Technology Laboratory, 1998.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[GLM02] H. Garavel, F. Lang, and R. Mateescu. An Overview of CADP 2001. *EASST Newsletter*, 4:13–24, 2002. CADP web page: http://www.inrialpes.fr/vasy/cadp.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[IT03a] P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.

[IT03b] P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. In *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 92–121. Springer, 2003.

[Lan05] F. Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In *Integrated Formal Methods (IFM'2005)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2005.

[MPR04] O. Maréchal, P. Poizat, and J.-C. Royer. Checking Asynchronously Communicating Components using Symbolic Transition Systems. In *Proc. of the International Symposium on Distributed Objects and Applications (DOA'2004)*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer, 2004.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[NM95] O. Nierstrasz and T. D. Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, 1995.

[OMG02] OMG. *CORBA Component Model (CCM)*. Object Management Group, 2002. Available at http://www.omg.org.

[OWL04] OWL Service Coalition. *OWL-S: Semantic Markup for Web Services*. The OWL Services Coalition, 2004. Available at http://www.daml.org/services.

[Poi06] Pascal Poizat. Eclipse Transition Systems. Deliverable of RNRT project STACS. `http://www.lami.univ-evry.fr/~poizat/documents/publications/Poi05.pdf`, 2006.

[Süd05] Mario Südholt. A Model of Components with Non-regular Protocols. In *Proc. of Software Composition (SC)*, volume 3628 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2005.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[W3C01] W3C. *Web Service Description Language (WSDL)*. World Wide Web Consortium, 2001. Available at `http://www.w3.org/TR/wsdl`.

[WHS01] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering, 2001.

[YS97] D.M. Yellin and R.E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.