Journal of Universal Computer Science, vol. 14, no. 13 (2008), 2182-2211
submitted: 31/10/07, accepted: 27/6/08, appeared: 1/7/08 © J.UCS

Composition and Run-time Adaptation of Mismatching Behavioural Interfaces

Javier Cámara
(Department of Computer Science, University of Málaga, Málaga, Spain
jcamara@lcc.uma.es)

Gwen Salaün
(Department of Computer Science, University of Málaga, Málaga, Spain
salaun@lcc.uma.es)

Carlos Canal
(Department of Computer Science, University of Málaga, Málaga, Spain
canal@lcc.uma.es)

Abstract: Reuse of software entities such as components or Web services raise composition issues since, most of the time, they present mismatches in their interfaces. These mismatches may appear at different interoperability levels: signature, behaviour, quality of service and semantics. The behavioural level is crucial and behavioural mismatches must all be corrected, although this is a difficult task. So far, most adaptation approaches which deal with behavioural mismatches work on a fixed description of components where all ports involved in their interfaces are known at design-time. Here, we focus on systems in which composition is affected by run-time behaviour of the system. This is the case in pervasive systems where a client interacts with a specific service by using new communication channels dynamically created. These are of special interest to allow private interaction between several entities.

In this article, we define a behavioural model inspired by the π-calculus to specify behavioural interfaces of components. Our model is particularly suitable for creating new channels dynamically, also taking concurrent behaviours into account. The dynamic nature of the systems we are dealing with obliges to apply adaptation at run-time, avoiding at the same time the costly generation of full descriptions of adaptors. The main contribution of this article is an adaptation engine that allows the dynamic creation of channels and applies at run-time a composition specification built at design-time. All the underlying formal foundations of our proposal have been implemented in a prototype tool that has been applied to system designs. Aspect-Oriented Programming has been studied as well, as a way to implement our engine for further application to real software components.

Key Words: Components, Behavioural Interfaces, Transition Systems, Mismatch, Composition, Run-Time Adaptation, Validation, Aspect-Oriented Programming.
Category: D.2, D.2.1, D.2.2, D.2.10, D.2.11, D.2.13

1 Introduction

Construction of software systems by reusing existing software entities[1] is a widely accepted process in the software engineering community. Thus, the design of an

[1] We will use *component* as a general term in the remainder of this article, standing for any kind of software entity (software component, Web service, agent, etc.)

application is mainly concerned with the selection, composition and adaptation of different pieces of software rather than with the programming of applications from scratch. Components are accessed through their public interfaces that usually distinguish four interoperability levels [Canal et al. 2006a]. The *signature level* provides operation names, type of arguments and return values, as well as exception types. The *behavioural level* specifies the order in which the component messages are exchanged with its environment. The *service level* groups other sources of mismatch, usually related to non-functional properties like temporal requirements, resources, security, etc. Finally, the *semantic level* is concerned about component functional specifications (*i.e.,* what they actually do).

Software components are seldom reusable *as is* because of mismatches that may appear at these different levels. These mismatches have to be corrected without modifying the component code due to its black-box nature. A very promising approach in software composition, namely *software adaptation*, aims at generating, as automatically as possible, *adaptor* components which are used to solve mismatches in a non-intrusive way [Yellin and Strom 1997, Canal et al. 2006a]. Adaptor generation approaches need an abstract specification (called *mapping* or *composition specification*) of the interactions and adaptations to be applied in order to make all the components work together correctly. Currently, industrial platforms only provide some means to describe components at their signature level (*e.g.,* CORBA's Interface Definition Language). However, most of the time mismatch occurs at the aforementioned behavioural level, due to an incompatibility in the order of the exchanged messages between components, which can lead to deadlocks (situations in which the system is blocked indefinitely). Similarly to several recent proposals in Component-Based Software Engineering and Software Adaptation [Alfaro and Henzinger 2001, Allen and Garlan 1997, Andrews et al. 2005, Arbab et al. 2002, Bracciali et al. 2005, Magee et al. 1999, Yellin and Strom 1997], we focus on the behavioural interoperability level, extending interfaces with a protocol description and dealing with the different compositional issues between them.

Many types of software systems are particularly characterized by structural changes at run-time, requiring communication along channels which are dynamically created. Such is the case of dynamic discovery and invocation of Web services, where entities may engage in the execution of the system at any time.

Existing adaptation approaches [Yellin and Strom 1997, Canal et al. 2006b, Schmidt and Reussner 2002, Inverardi and Tivoli 2003a, Bracciali et al. 2005] rely exclusively on static system information available at design-time (*i.e.*, a fixed structure of connected components and interfaces). Thus, they are not suitable to properly model and work out mismatch situations affected by run-time behaviour of the system. In order to assess the impact of such behaviour, we may consider a simple example based on the *Universal Description, Discov-*

ery and Integration (UDDI) [UDDI 2000] standard used in Web services, where: (i) a Web service requester searches an UDDI registry, finding the description of a desired service, (ii) the UDDI registry delivers a Web service description along with a reference to the Web service provider to the requester, and (iii) the requester connects to the Web service provider using the obtained reference to invoke the Web service.

The resolution of potential mismatch situations in such non-static systems entails a number of issues currently unaddressed in the aforementioned proposals. This work tackles these questions by making the following contributions:

- Formalisation of an adaptation model capable of handling the creation of new channel names, as well as their passing through channels. The need of such feature is put forward in our example by the fact that the Web service requester first has to interact with the UDDI registry, but later has to continue interaction with the service provider, unknown at design time, requiring new channels for this purpose.

- *Run-time adaptation engine* which aims at making components work together, correcting existing mismatches by using a composition specification. While adapting components dynamically, the execution of undesired behaviours leading to deadlock situations should be avoided. In fact, although a composition specification describes how to solve some mismatch cases that correspond to deadlocks, it is only an abstract description of how components work together, and therefore does not consider all the possible execution scenarios of the system. These can still include additional deadlock situations which are removed in approaches generating full descriptions of the adaptor [Inverardi and Tivoli 2003a, Canal et al. 2006b]. Similarly, our composition engine is capable of adapting a set of components at run-time and detect branches unable to terminate. This avoids deadlocking executions, and at the same time it does not require the costly generation of an adaptor for the whole system.

- New validation techniques to check that the composition specification makes the involved components work correctly as expected by the designer. These techniques are completely automated and based on our run-time adaptation engine.

A preliminary version of this work was published in [Cámara et al. 2007b], and is extended here in several aspects: (i) presentation of our proposal on an extended version of the case study, (ii) more detailed proofs for our algorithms, (iii) description of the Clint prototype tool, (iv) introduction of composition specification validation techniques,(v) implementation of our approach using Aspect-Oriented Programming, and (vi) an updated review and comparison with related work.

This article is structured as follows: Section 2 introduces our component model, and the case study we use throughout this article for illustration purposes. Section 3 presents the composition language which is used as an abstract description of how components interact, and how mismatches are worked out. Section 4 describes the engine which applies at run-time a composition specification. In Section 5, we present automated techniques to validate the composition specification, and some insights in order to help the designer to correct it in case this specification may induce erroneous executions. Section 6 describes the Clint prototype tool that implements our proposal for composition and run-time adaptation. In Section 7, we relate our proposal to programming languages and platforms. Aspect-Oriented Programming is a good candidate because it makes possible to dynamically intercept messages and apply adaptation as described in the composition specification. Section 8 compares our approach with related works. Section 9 ends the article with some concluding remarks.

2 Component Interfaces and Mismatch Detection

In this section, we introduce successively our component model based on Labelled Transition Systems (LTSs), and a discussion about mismatch detection.

2.1 Behavioural Interfaces

We choose to specify *behavioural interfaces* using LTSs as notation. This is very convenient for developing composition algorithms since they rely on the traversal of the different states of the components, and the transition function of LTS descriptions makes the access to the set of states and their connections straightforward. Messages involved in LTSs of components correspond to the operations used in its signature.

In this work, we extend the notion of LTS with *name passing*, such as the one found in the π-calculus. Thus, a component can receive as parameter of a message a new name that will be further used as a channel in the protocol. Such a feature is very useful since all channels are not always known at design-time, making the creation of new channels at run-time and the design of private interactions possible. Thus, a label is either an emission $a!(x)$, or a reception $a?(x)$ where x is a parameter used for name passing (parameters are facultative).

Another extension concerns the encoding of parallel process composition as a concurrent behaviour without communication since composition is expressed in a separate way (presented in Section 3). This is achieved by introducing *concurrent states*. These states are identified by two or more labels, one for each concurrent branch. They are similar to the *fork/join* states used in UML state diagrams, where more than one outgoing/incoming transitions encode concurrently executing branches of a process.

In the rest of this article, we refer to our extended LTS-based model as *par-LTS*.

Definition 1 par-LTS. A par-LTS is a tuple (A, S, I, F, T) where: A is an alphabet which corresponds to the set of labels hold by transitions, S is a set of states (either basic or concurrent), $I \in S$ is the initial state, $F \subseteq S$ are final states, and $T \subseteq S \times A \times S$ is the transition function.

In some cases, for conciseness reasons for example, a textual notation is better than a graphical one. Thus, a process algebra could be used as a higher level language to specify behavioural interfaces. A subset of the π-calculus (π-calculus without spawn operator) is a good candidate for that since it has at its disposal operators to express basic protocols but also concurrent behaviours and name passing (this is not the case in other process calculi such as CCS, CSP or LOTOS). Component par-LTSs can be automatically generated from processed expressed in a subset of the π-calculus using the operational rules of the process algebra (see [Milner 1999] for the Structural Operational Semantics of the π-calculus).

Regarding our model choice, most automaton-based formalisms such as *Hierarchical Automata* [Ramsokul and Sowmya 2006], *I/O-automata* [Lynch and Tuttle 1989], or *Symbolic Transition Systems* (STS) [Ingolfsdottir and Lin 2001] do not feature name passing, therefore they are not suitable for our purpose. To the best of our knowledge, the only automaton-based formalism available featuring name passing is *History Dependent Automata* [Montanari and Pistore 1997]. Although HD-Automata are adequate to represent the operational semantics of the π-calculus, we preferred an alternative and more expressive communication model where communication does not have to be given on the same channel names, and may involve more than two processes (n-ary *vs.* binary interaction).

Example: Rate Finder Service. Our running example consists of a service which finds the cheapest service rates (*e.g.,* calls, Internet access, etc.) for smart mobile phones from the different communication providers available, depending on the phone's location. Hence, when roaming, a mobile phone can connect to the service, and once the cheapest available rate has been received, the user can either request a connection to the provider or end its communication with the service if the offered rate is not interesting. Figure 1 presents the behavioural interfaces of the different components involved in this example.

The *Client* interface first requests the lowest rate to the service ($requestRate!$), and after receiving it, it can either ask to be connected to the cheapest provider's network (in this case it receives a private session identifier through $session?(sid)$), or else it can disconnect from the service ($end!$). If the

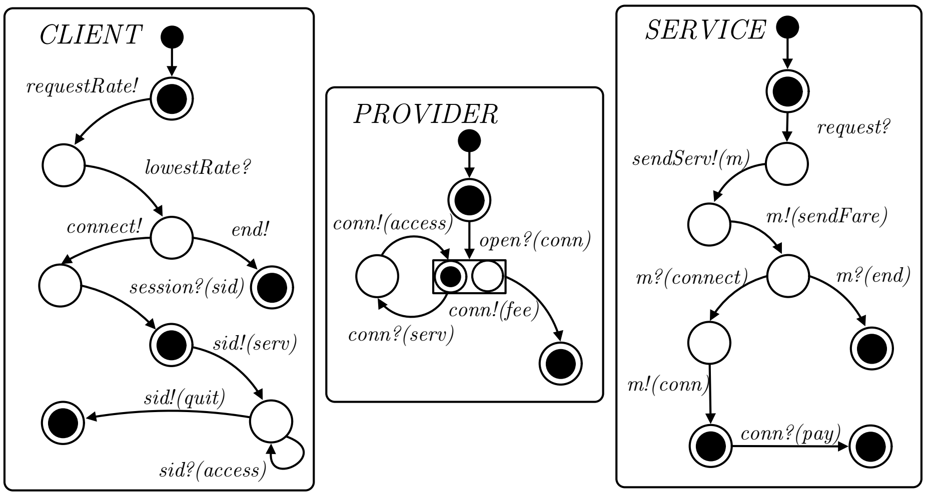

Figure 1: Par-LTSs for the rate finder service

client decides to connect, it interacts with the service provider through the aforementioned private session requesting a service, until it decides to exit from the provider's network.

The *Service* interface receives requests to find the provider with the lowest rates. Once the service receives a request, it serves this request on a new private communication channel m, which is returned to the client ($sendServ!(m)$). The requested information is returned ($m!(sendFare)$), and then the service waits for a connection request ($m?(connect)$), or else for the withdrawal from the service ($m?(end)$). If the client decides to connect to the network provider, then the service provides a private channel *conn* both to client and provider on which they can interact. Last, it may receive a payment ($conn?(pay)$) from the provider for which it has found a new client.

The *Provider* interface receives incoming client requests on a specific communication channel *conn* ($open?(conn)$), and then it can concurrently receive incoming service requests from the client, and pay a fee ($conn!(fee)$) to the finder service which has found a new client for him.

2.2 Mismatch Detection

Most of the time, components cannot be reused as they are because interactions among them would lead to an erroneous execution, namely a *mismatch*. In practice, mismatch situations may be caused when message names do not correspond, the order of messages is not respected, a message in one component has no counterpart, or a message matches with several messages.

More formally, cases of mismatch lead the whole system to deadlock states. A deadlock state is a state which is not final and for which no outgoing transitions exist. A system deadlocks when all its constituent components are blocked because at least one of them is in a deadlock state. Accordingly, mismatch detection is figured out in two steps: (i) computing the full system using the par-LTS product where (binary) communication is enforced on the same name of messages, (ii) searching for deadlock states in the resulting par-LTS.

However, this method does not extract all the mismatch cases but only those that can be reached assuming that the involved components can interact using same names of messages. This test can then be used in a first step to start the construction of the composition specification (see Section 3) that describes how mismatch situations are solved. While building this specification, validation techniques presented in Section 5 will help the designer in a second step to build incrementally and correct the composition specification. To the best of our knowledge, there are no techniques so far that allow to detect all the mismatches between a set of components to be assembled. Accordingly, beyond the test mentioned above that basically is reduced to indicate if protocols are compatible or not, the only way to extract all the mismatches is (visual) analysis performed by the designer.

Example. The composition of the different components in this example is subject to different mismatch situations:

(M1) *Name mismatch* can occur if a particular process is expecting a particular input event, and receives one with a different name (*e.g.*, $Client$ sends $requestRate!$ while $Service$ is expecting $request?$).

(M2) *Independent evolution* is given if an event on a particular interface has not an equivalent in its counterpart's interface. If we take a closer look at the $Client$ and $Service$ interfaces, it can be observed that while the client is expecting the lowest rate just after its request, the service is sending two messages ($sendServ!(m)$ and $m!(sendFare)$). While the latter actually sends the lowest rate to the client, the former has no correspondence on the $Client$ interface.

(M3) *Broadcasting.* It can be required by the nature of the composition to synchronise more than one component on a single communication. In our case study this can be observed when the service sends a private name ($m!(conn)$) to the provider and the client, enabling them to connect directly. In this case, the $Provider$ interface receives an incoming connection ($open?(conn)$), although the $Client$ needs to be notified that it has been connected to the provider as well ($session?(sid)$).

(M4) *Private vs. non-private communication.* New processes can be created in order to serve particular requests from incoming clients through private communication channels. However, different components may be built with different communication protocols. Particularly, in our case study we can observe that while the *Client* interface is expecting non-private communication through $lowestRate?$ in order to receive the lowest available rate, the *Service* interface creates instead a private communication channel which is going to be used to send this information ($m!(sendFare)$), resulting in a mismatch.

3 Composition and Adaptation Language

In this section, we present our composition language that makes communication between components explicit, and specifies how to work out mismatch situations.

To make communication explicit, we choose *synchronisation* (or *synchronous*) *vectors*, which denote communication between several components, where each event appearing in one vector is executed by one component and the overall result corresponds to a synchronisation between all the involved components. A vector may involve any number of components and does not require interactions on the same names of events as it is the case in process algebra. Vectors can describe expressive communication patterns, such as broadcast communication (one sender and several receivers), or synchronisation between several senders and several receivers (in this case, emitted names appearing as parameter have to match).

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

To identify component messages in a vector, their names are prefixed by the component identifier, *e.g.*, $\langle c_1 : comm!, c_2 : \varepsilon, c_3 : comm? \rangle$.

We use as abstract notation for our composition language an LTS with vectors on transitions. This LTS is used as a guide in the application order of interactions denoted by vectors. This order between vectors is essential in some situations in which mismatch can be avoided by applying some vectors in a specific order. If only correpondences are necessary between components, the vector LTS may let abstract the vector application order by including a single state with all vector transitions looping on it.

Definition 3 Composition Language. A composition language for a set of components C_i, $i \in \{1, .., n\}$, is a couple (V, C_{ce}) where V is a set of vectors for components C_i, and C_{ce} is a vector LTS.

Reordering of messages is needed in some communication scenarios to ensure a correct interaction when two communicating entities have messages which are not ordered as required. In our proposal, such a reordering of messages can be specified making it explicit in the composition specification. Let us consider two components C_1 and C_2 (written in CCS for conciseness), exchanging login and request information, with messages that have to be reordered to make the communication possible: $C_1 = log!.req!.0$, $C_2 = query?.id?.0$. Our approach can reorder messages using the following vectors $\langle c_1 : log!, c_2 : \varepsilon\rangle$, $\langle c_1 : req!, c_2 : query?\rangle$, and $\langle c_1 : \varepsilon, c_2 : id?\rangle$, in which we specify that the interaction on *log* is desynchronised, temporarily memorised until its use for effective interaction on *id*. This leads to an execution trace where components C_1 and C_2 execute successively the following messages: $c_1\!:\!log!$, $c_1\!:\!req!$, $c_2\!:\!query?$, $c_2\!:\!id?$.

Example. Considering our running example, we propose a composition expression in order to solve the different mismatch situations described in the previous section. This is specified by the following set of synchronisation vectors, and the vector LTS depicted in Figure 2. The events in the vectors are prefixed with c, s, and p, which stand for the *Client*, *Service*, and *Provider* components, respectively. In addition, only formal channel names are used for private communication, since actual channel names are not available until the different components are being composed:

$$
\begin{aligned}
&v_{req} = \langle c\!:\!requestRate!, s\!:\!request?, p\!:\!\varepsilon\rangle\\
&v_{sserv} = \langle c\!:\!\varepsilon, s\!:\!sendServ!(m), p\!:\!\varepsilon\rangle\\
&v_{far} = \langle c\!:\!lowestRate?, s\!:\!m!(sendFare), p\!:\!\varepsilon\rangle\\
&v_{conn} = \langle c\!:\!connect!, s\!:\!m?(connect), p\!:\!\varepsilon\rangle\\
&v_{sess} = \langle c\!:\!session?(sid), s\!:\!m!(conn), p\!:\!open?(conn)\rangle\\
&v_{end} = \langle c\!:\!end!, s\!:\!m?(end), p\!:\!\varepsilon\rangle\\
&v_{serv} = \langle c\!:\!sid!(serv), s\!:\!\varepsilon, p\!:\!conn?(serv)\rangle\\
&v_{nserv} = \langle c\!:\!\varepsilon, s\!:\!\varepsilon, p\!:\!conn?(serv)\rangle\\
&v_{acc} = \langle c\!:\!sid?(access), s\!:\!\varepsilon, p\!:\!conn!(access)\rangle\\
&v_{quit} = \langle c\!:\!sid!(quit), s\!:\!\varepsilon, p\!:\!\varepsilon\rangle\\
&v_{fee} = \langle c\!:\!\varepsilon, s\!:\!conn?(pay), p\!:\!conn!(fee)\rangle
\end{aligned}
$$

First, while v_{req} for instance deals with name mismatch (M1), v_{sserv} and v_{quit} make events correspond to none, solving the different independent evolution situations (M2). Next, v_{sess} broadcasts the emission of $m!(conn)$ both to *Client* and *Provider* components to establish a connection between them (M3). Finally, v_{far} solves a private communication issue by making the client receive on *lowestRate*? the fare sent by the service through channel m (M4).

The writing of the vector LTS for the composition specification (see Figure 2) is simplified by keeping in some places non-determinism in the composition specification, for instance by leaving open different possible application orders of v_{req}, v_{conn}, v_{sserv}, and v_{far} to let the composition process decide which order – if any – is correct. On the contrary, the order of vector application may be enforced to make the different evolutions of the system explicit. Note that in some cases,

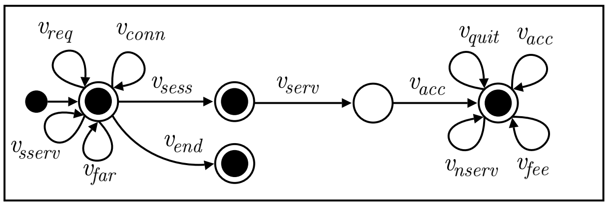

Figure 2: Case study vector LTS

this order is mandatory to make the system work as required and avoid undesired behaviour. A classic example is when one component, a client for example, works in connected mode (one connection and multiple access to the service) whereas the server component requires a connection each time the service is accessed [Canal et al. 2006b] (non-connected mode). In this case v_{nserv} must be explicitly applied after v_{serv} and v_{acc} in order to allow the client to receive subsequent service access, otherwise the system would be able to evolve only if the client quits the provider, receiving only a single service access.

4 Run-time Adaptation

Our run-time adaptation engine coordinates all the components involved in the system with respect to a set of interactions defined in the composition specification. Consequently, all the components are communicating through the engine [Inverardi and Tivoli 2003b]. Imagine two components: $C_1 = on!.0$ and $C_2 = activate?.0$ with vector $\langle c_1 : on!, c_2 : activate? \rangle$ as a solution to these mismatching messages. The engine will communicate first with component C_1 through $on?$, and then will interact with component C_2 through $activate!$. The engine synchronises with the components using the same name of messages but using reversed directions, *e.g.,* communication between *on!* in C_1 and *on?* in the engine. Furthermore, the engine always starts a set of interactions formalised in a vector by the receptions (*on?*), and next handles the emissions (*activate!*): it would be meaningless to send something that has not been received yet.

This section introduces successively two algorithms. The first one searches for the existence of a final state for the global system using depth-first search. The second one applies a composition specification at run-time resulting in the dynamic execution of the involved components.

4.1 Existence of Global Final States

Although the composition specification aims at solving mismatch cases that correspond to deadlocks, its application can still lead to remaining deadlocks.

Indeed, the composition specification is an abstract description of how components work together, and does not take into account all the possible execution scenarios of the system. Removing these remaining spurious interactions is required to let the system reach a final state. Since the composition specification is applied at run-time, it is not possible to apply the removal of deadlocks as a pre-processing as it is the case in static coordination and adaptation approaches [Inverardi and Tivoli 2003a, Canal et al. 2006b]. Therefore, before applying a vector which belongs to the composition specification, we check that after the application of this vector, there exists at least one global final state for the whole system (*i.e.*, all components and the composition specification are in a final state). Thus, our adaptation engine will prevent the system to end up in deadlocking situations. We will illustrate that on a specific situation which takes place in our running example further in this section.

Algorithm 1 takes as input the component par-LTSs C_i, the composition specification C_{ce}, a vector v, and the current state of the system (*i.e.*, the current *states* of the components as well as the current state s_{ce} of the composition specification). This algorithm relies on a depth-first search traversal, and stops as soon as a final state for the whole system has been found. The main idea is that vectors belonging to the composition specification are applied going in depth until we reach either a final state (end of the algorithm), or a deadlock state. In the latter case, we backtrack and try another path. We keep track of the already traversed states to avoid endless execution of our algorithm.

Now, we define more formally the different functions used in Algorithm 1. In order to check if all the components and the composition specification have reached their final state, we define the function *final* as:

$$final(states, F_i, s_{ce}, F_{ce}) =$$
$$f(states[1], F_1) \wedge \ldots \wedge f(states[n], F_n) \wedge s_{ce} \in F_{ce}$$
$$f([s_1, \ldots, s_m], F) = s_1 \in F \wedge \ldots \wedge s_m \in F$$

Function *applicable* returns an interaction vector extracted from V. In case several vectors can be applied, a single one is non-deterministically chosen.

$$applicable(states, C_i, s_{ce}, C_{ce}) =$$
$$\begin{cases} v & \text{if } (s_{ce}, v, s'_{ce}) \in T_{ce}, (\forall l \in v)\ l \neq \varepsilon, j \in \{1, \ldots, n\}, s_j = states[j], \\ & run(s_j, l, T_j) = s'_j, \text{ and } s'_j \neq s_\perp \\ v_\perp & \text{otherwise (no vector applicable)} \end{cases}$$

The communication case in which vector v involves several emissions can be handled enforcing in the function *applicable* that all the names sent in the emissions are the same: $par(l_1) = \ldots = par(l_k)$ where $\{l_1, \ldots, l_k\} = extract\_emissions(v)$, function *extract_emissions* retrieves all the emissions which belong to a specific vector: $extract\_emissions(\langle l_1, \ldots, l_n \rangle) = \{l_i \mid l_i = e!(x)\}$ and function *par* retrieves sent or received names: $par(e!(x)) = x, par(r?(x)) = x$.

Algorithm 1 *exist_final*

tests if the application of a vector may lead to a final state for the system
inputs *states*, compo. $C_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, s_{ce}, $C_{ce} = (A_{ce}, S_{ce}, I_{ce}, F_{ce}, T_{ce})$, and a vector v
output a boolean

$visited := \{\}$ *// set of visited states*
$current := s_{ce}$ *// current state in the composition specification*
$path := []$ *// current path*
$cloop := true$ *// loop condition*
$cypaths := \{\}$ *// cyclic paths*
while $\neg final(states, F_i, s_{ce}, F_{ce}) \wedge cloop$ **do**
  $states := next\_states(v, states, T_i)$
  $current := next(current, v, T_{ce})$
  $path := append(path, (current, states))$
  $visited := visited \cup \{current\}$
  while $\neg final(states, F_i, s_{ce}, F_{ce}) \wedge v = v_\perp \wedge path \neq []$ **do**
    $v := applicable(states, C_i, current, C_{ce})$
    $states' := next\_states(v, states, T_i)$
    $current' := next(current, v, T_{ce})$
    $path' := append(path, (current', states'))$
    $cond := next(current, v, T_{ce}) \in visited \wedge stability(path', cypaths)$
    if $cond \wedge \neg isin(path, cypaths)$ **then**
      $cypaths := cypaths \cup \{path\}$
    end if
    if $v = v_\perp \vee cond$ **then**
      $current := left(last(path))$
      $states := right(last(path))$
      $path := remove\_last(path)$
    end if
  end while
  $cloop := (v \neq v_\perp)$
end while
$return\ final(states, F_i, s_{ce}, F_{ce})$

Function *next_states* computes the next states of the involved components from their current states and a vector:

$next\_states(\langle l_1, \ldots, l_n\rangle, states, T_i) = [s'_1, \ldots, s'_n]$
where $\forall i \in \{1, \ldots, n\}\ run(states[i], l_i, T_i) = s'_i$

Function *next* computes the next state in the composition specification being given a state s and a vector label v:

$next(s, v, T) = s'$ where $(s, v, s') \in T$

In the inside *while* loop, we rely on a stability function (*stability*) that enables the application of a vector even if it leads to an already traversed state of the composition specification. This is necessary since in some cases, several executions of a cycle in the vector LTS may make all the involved components finally lead to a termination state. However, we have to control the exploration of these paths to avoid non-terminating executions of our algorithm. Accordingly, this stability function returns *true* (*false* otherwise) if a looping behaviour is detected in the current path (repetition of same vectors and traversal of same states in component par-LTSs). Next, the algorithm backtracks to explore an unvisited part of the system. Cyclic paths are easily detected by analysing the *path* variable obtained after the application of a new vector, and several existing algorithms can be used to find out these cycles. In this article, we chose the Floyd's cycle-finding algorithm [Knuth 1969][2]. To avoid to traverse again these cyclic paths, we store them in a variable *cypaths*, and function *stability* forbids to explore them again.

$$stability(path, cypaths) = \begin{cases} true & \text{if } isin(path, cypaths) \\ find\_cycle(path) & \text{otherwise} \end{cases}$$

where algorithm $find\_cycle$ returns *true* if a cycle is detected in the current path, or *false* otherwise.

Let us formalise the set of functions we apply on the *path* variable that corresponds to a list of couples:

$append([c_1, \ldots, c_n], c) = [c_1, \ldots, c_n, c]$
$last([c_1, \ldots, c_n]) = c_n$
$remove\_last([c_1, \ldots, c_{n-1}, c_n]) = [c_1, \ldots, c_{n-1}]$

Functions $left$ and $right$ apply to a couple and are defined as: $left((c_l, c_r)) = c_l$, $right((c_l, c_r)) = c_r$. Last, function *isin* tests if a path appears as subpath in the list of cyclic paths.

$isin(p, \{p_1, \ldots, p_n\}) = isin'(p, p_1) \vee \ldots \vee isin'(p, p_n)$

$$isin'([c_1, \ldots, c_m], [c'_1, \ldots, c'_k]) = \begin{cases} c_1[1] = c'_1[1] \wedge \ldots \wedge c_m[1] = c'_m[1] & \text{if } m \leq k \\ false & \text{otherwise} \end{cases}$$

The complexity of Algorithm 1 is linear – O($|S_{ce}|$) – since, in the worst case, the whole vector LTS standing for the composition specification is traversed, the cycle-finding algorithm is applied during this traversal, and this algorithm obliges to traverse again the vector LTS at most a constant number of times. For each vector transition in the composition specification, component par-LTSs are not traversed because we keep track in the *states* variable of the current state of each component par-LTS, therefore, we can access directly their next states.

Theorem 4 Algorithm 1 Termination. *Given n component par-LTSs C_i, a composition specification C_{ce}, the current state of the system (components and composition specification), and a vector to be applied, the algorithm terminates.*

Proof (intuition). The algorithm runs until (i) it has found a global final state, or (ii) it has traversed the vector LTS completely without finding a final state.

[2] The authors quote a Knuth's reference because Floyd has never published his cycle-finding algorithm. Knuth was the first to present it in the referenced book, and acknowledged Floyd for the algorithm.

First of all, let us comment more precisely conditions of both *while* loops. The outer loop stops when the global state is a final state, or no vector is applicable. The inner loop ends when an applicable vector has been encountered. The *stability* function allows to apply a vector that can be run again although all the states have already been visited. Indeed, in some cases, some part of the system (components and composition specification) may loop several times on same interactions before the whole system reaches a global final state. To avoid endless executions, the *stability* function relies on a cycle-finding algorithm which is used to detect repeated behaviours, and makes the algorithm terminate when such cycles are found out during the traversal.

Thus, the first case (i) is ensured thanks to both loop conditions that rely on the *final* predicate: as soon as a global final state is reached, both conditions turn out to be false, and the algorithm terminates.

The second case (ii) is ensured by a joint use of the *visited* variable and the *stability* function. The *visited* variable stores the set of already traversed states and is used within the inner *while* loop to avoid the firing of a new vector that would lead to an already visited state. However, some states must be traversed several times and some vector transitions run again to reach some final states. This is achieved using the *stability* function that permits such a repeated application while applying a cycle-finding algorithm to detect looping behaviours. In the latter case, these cyclic paths are stored in the *cypaths* variable not to treat them again if they have already been traversed. □

Theorem 5 Algorithm 1 Correctness. *Given n component par-LTSs C_i, a composition specification C_{ce}, the current state of the system (components and composition specification), and a vector to be applied, the algorithm returns a boolean indicating the existence (or not) of a global final state for the system.*

Proof (intuition). We start pointing out that our algorithm is based on classic graph traversal algorithm, but extended to deal with our precise problem (composition of components using several LTSs). This proof has to deal with two cases. In the first case, if it exists a final global state, this state will be encountered, and the algorithm will stop returning *true*. In the second case, there is no final state, and the algorithm returns *false*. The first case is the most relevant since in the second case, the single important point is termination, and it was already discussed in the proof presented above.

The first case relies on a depth-first search algorithm with backtracking in case no vector is applicable in the current state of the overall system. Accordingly, all the states of the system are visited. The *stability* function allows to traverse again visited states if an applicable vector exists up to cycles that would make the system loop forever. Finally, when a final state is encountered, conditions in both loops make the algorithm terminate, and the result of predicate $final$ is returned, that is *true* in this case. □

Example. We focus on a piece of our running example to illustrate how the final state detection is used to avoid a deadlock situation. In Figure 3, we can observe that the highlighted part of the client specification first requests a service, then can access it as many times as needed, and at some point quits. On the other hand, the provider waits for a request and provides a service.

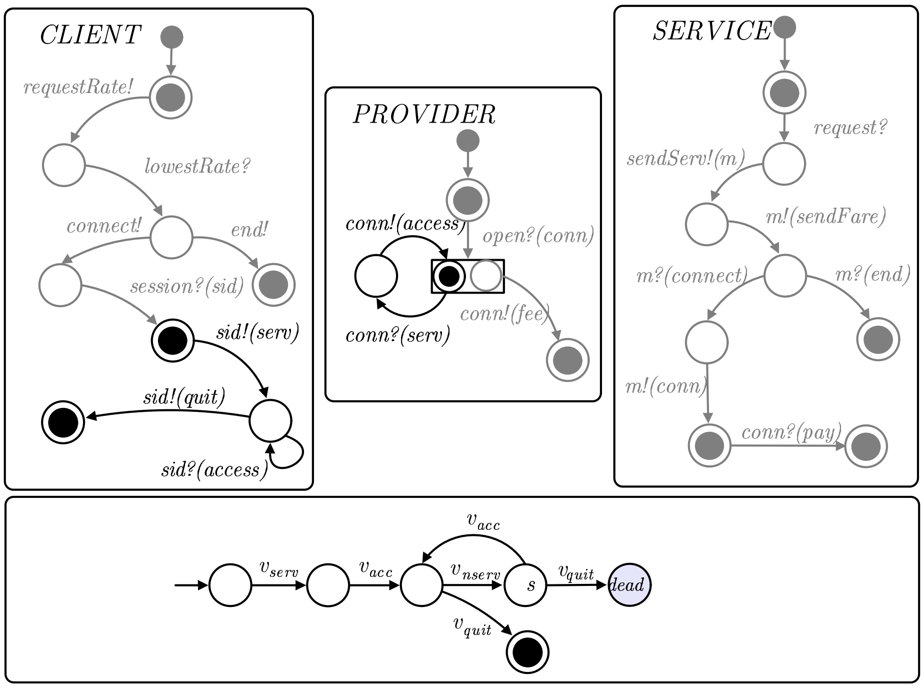

Figure 3: Deadlocking application of vectors

Vectors v_{serv}, v_{acc}, v_{nserv}, v_{quit} in Figure 2 are those dedicated to this part of the composition. We show in Figure 3 the transition system summarising all the possible applications of these vectors with respect to the highlighted parts of the interfaces. After the application of v_{nserv} (state s), the client may evolve by v_{quit} (this would be a correct evolution of the client to its final state) which would insert a deadlock in the system since the provider would be eventually blocked: it can send an access $conn!(access)$ but the composition expression has reached a final state, and no vectors are fireable. Fortunately, our *exist_final* algorithm detects that the application of v_{quit} in state s would lead to a deadlock (no future correct final state), and therefore the only vector which can be run is v_{acc}.

4.2 Run-time Adaptation Engine

This section presents our adaptation engine which applies a composition specification at run-time since new channel names are created dynamically and cannot be known beforehand. The right substitutions of channel names are dynamically performed by the algorithm. We will illustrate how these substitutions are essential to make the communications between the engine and the involved components work.

Algorithm 2 manages the composition between several component par-LTSs with respect to a given composition specification. We propose a composition which respects the sequential interactions described within vectors of the composition specification, that is, events belonging to two different vector transitions of the composition specification are never interleaved. Such an interleaving may happen in some cases in which components involved in two subsequent vectors are completely unrelated.

The composition algorithm applies successively vectors that can be fired with respect to the current state of the system. For each vector, first receptions in the adaptation engine are executed (corresponding to emissions in the components), and then emissions (corresponding to receptions in the components). The algorithm keeps track on the current state of the vector LTS (s_{ce}) as well as on the current states of the components ($states$). The algorithm ends when the global system has reached a final state. Since the selection of an applicable vector also relies on the final state existence algorithm presented in Section 4.1, we engage the first time in the *while* loop only if there exists a global final state for the system ($v \neq v_\perp$), otherwise the composition is not launched. Notice the abstract *loop* condition introduced in order to control iterative behaviours within the system at hand. This condition can be adjusted to make the algorithm continue the composition beyond a global termination state an arbitrary number of times (if there are applicable vectors available). We use a substitution environment E for each of the participating components, defined as a function where each formal name is associated to one actual name. Such an environment is useful to replace formal names used in the vectors by real ones that are generated by the components during their execution.

Let us define the functions used in the composition algorithm. Function *select_vector* bases on the *applicable* and *exist_final* functions, and is in charge of selecting a vector that applies and ensures the existence of a future final state:

$$select\_vector(states, C_i, s_{ce}, C_{ce}, E_i) =$$
$$\begin{cases} v & \text{if } v = applicable(states, C_i, s_{ce}, C_{ce}, E_i), \\ & v \neq v_\perp,\ exist\_final(states, C_i, s_{ce}, C_{ce}, v) \\ v_\perp & \text{otherwise (no vector applicable)} \end{cases}$$

We define *emissions* and *receptions* which return the set of emissions and receptions respectively, of any given synchronisation vector. These functions also

Algorithm 2 *run-time_composition*

composes at run-time a set of components wrt. a composition specification
inputs components $C_{i\in\{1,\dots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, compo. $C_{ce} = (A_{ce}, S_{ce}, I_{ce}, F_{ce}, T_{ce})$

$states := [[I_1], \dots, [I_n]]$ // *current states in* C_i
$E_1 := \emptyset, \dots, E_n := \emptyset$ // *substitution environments*
$s_{ce} := I_{ce}$ // *current state in the composition specification*
$v := select\_vector(states, C_i, s_{ce}, C_{ce}, E_i)$
while $v \neq v_\perp \ \wedge\ (\neg final(states, F_i, s_{ce}, F_{ce}) \ \vee\ loop)$ **do**
  $Rec := emissions(v, E_i)$
  $Em := receptions(v, E_i)$
  $x := generateVar()$
  repeat // *effective receptions*
    $r?(x) \mid r \in Rec, j \in \{1, \dots, n\}, s_j = states[j]$,
    $l \in A_j, run(s_j, l, T_j) = s'_j, r = obs(l, E_j)$
    $Rec := Rec\backslash\{r\}$
    $states[j] := s'_j$
  until $Rec = \emptyset$
  repeat // *effective emissions*
    $e!(x) \mid e \in Em, j \in \{1, \dots, n\}, s_j = states[j]$,
    $l \in A_j, run(s_j, l, T_j) = s'_j, e = obs(l, E_j)$
    $Em := Em\backslash\{e\}$
    $states[j] := s'_j$
    $E_j := E_j \cup \{(par(l) \mapsto \$x)\}$ // *\$x is the value of var. x*
  until $Em = \emptyset$
  $s_{ce} := next(s_{ce}, v, T_{ce})$
  $v := select\_vector(states, C_i, s_{ce}, C_{ce}, E)$
end while

substitute formal names used in the vectors by the actual ones appearing in the environment E:

$$emissions(\langle l_1, \dots, l_n\rangle, E_i) = em(l_1, E_1) \cup \dots \cup em(l_n, E_n)$$

$$em(l, E) = \begin{cases} \{sub_n(e, E)\} & \text{if } l = e!(x) \\ \emptyset & \text{if } l = r?(x) \vee l = \varepsilon \end{cases}$$

$$receptions(\langle l_1, \dots, l_n\rangle, E_i) = rec(l_1, E_1) \cup \dots \cup rec(l_n, E_n)$$

$$rec(l, E) = \begin{cases} \{sub_n(r, E)\} & \text{if } l = r?(x) \\ \emptyset & \text{if } l = e!(x) \vee l = \varepsilon \end{cases}$$

where

$$sub_n(e, E) = \begin{cases} E(e) & \text{if } e \in dom(E) \\ e & \text{otherwise} \end{cases}$$

The *observational part* of an event l is defined as: $obs(e!(x), E) = sub_n(e, E)$, $obs(r?(x), E) = sub_n(r, E)$. The $\backslash$ operator denotes the set difference defined as: $E_1\backslash E_2 = \{x \mid x \in E_1 \wedge x \notin E_2\ \}$. The abstract function $generateVar$ derives new variables when required.

The complexity of Algorithm 2 is polynomial – $O(|S_{ce}|^2)$. For each composition step, we know the current state of the whole system, therefore the extraction of applicable vectors with respect to the current state of the whole system is straightforward. However, for each vector, the function *exist_final* is called, and in the worst case, the vector LTS is traversed a constant number of times (see complexity of *exist_final* algorithm in Section 4.1).

Theorem 6 Algorithm 2 Correctness. *Given n component par-LTSs C_i, and a composition specification C_{ce}, the algorithm makes all of them (C_i and C_{ce}) terminate in their respective final states.*

For this algorithm, correcteness ensures also termination, consequently we present a single proof.

Proof (intuition). In a first case, the algorithm does not start any interaction since there is no reachable global final state, and the *while* loop is never entered. In such a case, all the components do not evolve and they end in their initial states. We have assumed initial states being final in behavioural interfaces of components to ensure the correctness of the composition. Composition being a process which is dependent of the composition specification, the process may yield a system in which no interactions are possible in case of having an incorrect or under-specified composition specification. Tagging all initial states as final ensures correctness.

In the second case, the algorithm makes all the components terminate in one of their final states as well as the composition specification. This second situation corresponds to the firing of the *while* loop at least once. In this case, it means that it exists at least one final state for the global system. The successive selection of vectors that can be applied will avoid all the possible deadlock situations (thanks to the *exist_final* algorithm), and therefore the algorithm will make all the components converge to one of these final states. The *while* loop ends when one global final state is reached. □

Example. Figure 4 gives a possible execution scenario that is obtained running the adaptation engine on our example. The figure shows interactions that occur in the system, vectors that are executed, and extensions of the environments. Although this figure corresponds to a specific scenario, we have added in dashed lines some other possible evolutions of the system. Similarly, the execution might have stopped at the mid-bottom state of the figure without any interaction between the client and the provider (because in this state, all the components are in final states, and the composition expression as well), or the execution might have continued after the top left final state in case the client would have required another service to the provider component.

As far as the firing of events is concerned, we recall that the adaptation engine is coordinating the whole system, therefore all the messages are canalised through it. As an example, when the first vector is run, it corresponds to two

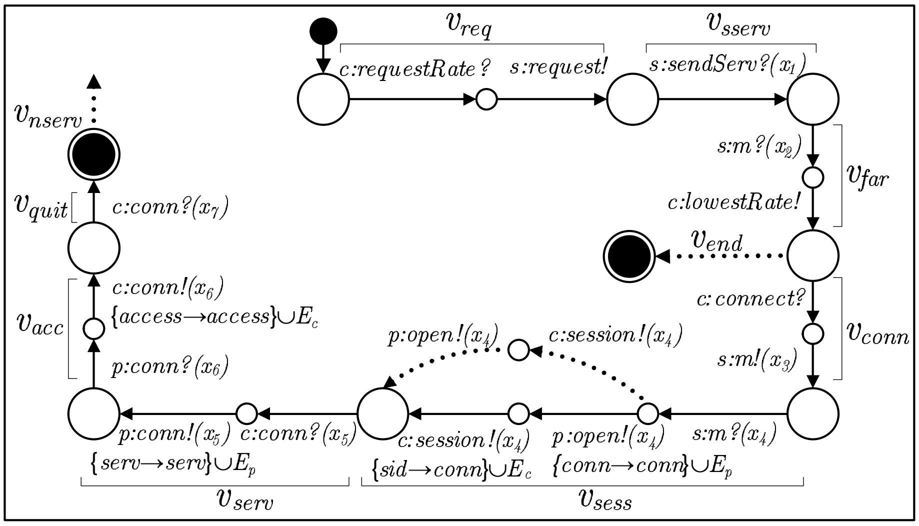

Figure 4: Run-time composition in action

interactions, the first one ($requestRate$) between the client and the engine, and another one ($request$) between the engine and the server.

During execution, the different environments (E_c, E_s, and E_p) for the $Client$, $Service$, and $Provider$ components may be updated. For instance, focusing on the reception of x_4 coming with message m from the service component (firing of vector v_{sess} in the mid-bottom of Figure 4), it can be observed that environments E_c and E_p are extended associating their formal names (sid and $conn$, respectively) which appear in their interfaces and in the vector by the actual one which is received in x_4, namely $conn$. Environments are also necessary to make interactions between the engine and the components work correctly. For example, while applying vector v_{serv}, the formal name sid in $sid!(serv)$ is replaced using the actual name $conn$. This correspondence is achieved using the couple $(sid, conn)$ in E_c, assigned while executing v_{sess}.

In some cases, the engine receives a value, without forwarding it (see vector v_{sserv} in the top part of Figure 4). Likewise, a vector may express a correspondence between a message without parameter, and a message with one (*e.g.,* v_{conn}). In this case, the engine receives no value, and emits an empty value x_3 (see the right hand part of the figure). Finally, as regards the 3-party communication involved in vector v_{sess} (bottom of Figure 4), we show in dashed lines that since the engine is sending two messages, both orders are possible.

5 Validation of Composition Specifications

A composition specification may induce erroneous executions of the system. Indeed, this specification has to be manually written by a designer who can introduce errors while making explicit interactions between the different components at hand. This specification task is error-prone also because of the complexity of the system that the designer has to deal with. To help the designer in this task recent proposals aimed at building adaptors, and therefore adaptation specifications, incrementally [Poizat and Salaün 2007]. Other works [Ben Mokhtar et al. 2005, Brogi et al. 2006a] focus on semantic approaches to automatically build such compositions. However, these latter approaches assume specific descriptions of component interfaces, and in particular that a semantic description of components or (Web) services is provided.

In this section, we propose some new validation techniques to check that the composition specification makes the involved components work correctly and as expected by the designer. These techniques are completely automated, and are based on the composition engine we have presented in Section 4. The basic idea is to generate many execution traces using our engine that we will use in a second step to evaluate the composition specification. In order to obtain all possible execution traces, and above all the erroneous ones, the final state existence test is turned off. From such a set of traces, we extract the following information that can be used by the designer to refine and if necessary correct the composition specification:

- *Unreachable states* allow the designer to identify which states of the composition cannot be visited. This is specially interesting in the case of final states.
- *Unreachable transitions* identify the transitions which cannot ever be fired, preventing access in some cases to a specific state or branch of the composition specification.
- *Deadlock traces* are particular sequences of applied vectors that lead to a deadlock situation. This information is not obvious at all and the potential number of vector sequences to apply is usually huge. In non-trivial cases, it is impossible for the engineer to check all these potential deadlock situations manually.

6 Tool Support and Applications

The approach for run-time composition and adaptation that we have presented in this article has been implemented in a prototype tool, called Clint (Composition Language Interpreter). Clint implements the algorithms that simulate run-time adapted interaction on a set of components with respect to a composition specification given as a set of vectors and a vector LTS. Clint is capable of generating a graphical representation of the different inputs (behavioural interfaces and composition specification), and help the user to explore the composition through an interactive simulation by visualising the evolution of the system step-by-step. In addition, the user has the possibility of validating the composition specification, visualising errors which are highlighted on the graphical representation of the vector LTS.

In order to perform the interactive simulation of the composition, the tool offers the set of applicable vectors to the user at each step of the composition (listed on the *Applicable Vectors Panel*, located at the upper-left corner of Figure 5). Then, the user selects a vector which is fired, updating the state of the system, both in the *Simulation Trace Panel*, which outputs a textual representation of the simulation trace (bottom-left corner), and the graphical representation of the interfaces and vector LTS. There are two different modes for interactive simulation:

– Safe mode (default). The tool offers the user only *safe* vectors for selection (*i.e.,* vectors which do not lead to a deadlock state of the system).

– Unsafe mode. The interface offers all applicable vectors to the user. Although this allows the application of vectors leading to deadlock states, this possibility may be interesting in order to observe and understand potential flaws in the current composition specification.

The tool is also able to perform validation on the composition specification *wrt.* a set of component par-LTSs. In order to achieve this, many random execution traces are automatically generated using the composition engine (unsafe mode). From such a set of traces, states and transitions are labelled with the overall number of times they are traversed. These traces are also used to identify unreachable states or transitions which are never traversed in the composition specification. Clint colours the composition specification, highlighting such unreachable states and transitions in the graphical representation. Moreover, traces leading to deadlock situations can be identified by the tool. When a non-final state in the vector LTS is reached, and no further vectors can be applied, the tool classifies the sequence of vectors which has led to the current situation as a deadlock trace.

Specifically, Figure 5 features a modified version of the composition specification used for our rate finder service. We can observe that instead of a transition on v_{acc} going from $R3$ to $R4$, we have inserted a transition on v_{fee}. The application of v_{fee} before reaching $R4$ prevents the firing of that same vector once the composition has reached state $R4$. Hence, the transition of v_{fee} in $R4$ is highlighted in red on the graphical representation since it cannot ever be fired.

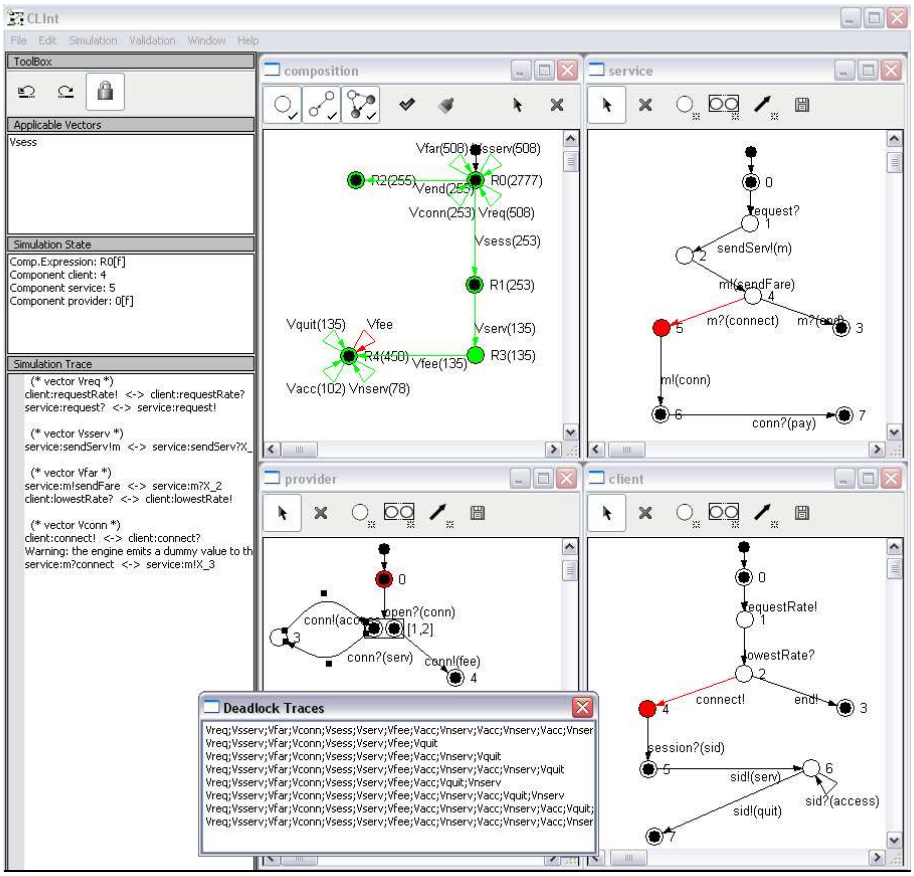

Figure 5: Validation of the rate finder service using Clint

Likewise, the tool is able to identify traces leading to deadlock situations. Particularly, we can observe in Figure 5 that all traces which involve a bad order in the application of vectors (as described in Section 4.1) are displayed on the user interface of the tool.

Clint has been implemented in Python, using the wxWidgets toolkit technology for the development of the user interface and the Object Graphics Library (OGL) to visualise the different inputs of the tool. Interfaces are described using the `.aut` textual format [Garavel et al. 2007], whereas the composition specification is written in an XML-based format specific to the tool.

Clint has been validated on about 200 examples, which range from small ones to experiment boundary cases, to real-world examples such as a travel agency, push-out advertisements, multi-device service, rate finder service, on-line computer material store, Video-on-demand, music player, disk manager, library management systems, a SQL server and several other client/server systems. In our validation base, we distinguished two sets of examples. The first set (approx. 120) concerns systems that do not involve name passing. These examples were very useful to experiment our run-time composition algorithms, and the global final state search. The second set (approx. 80) focuses on systems that require name passing and concurrent behaviours.

We use three examples of different sizes from our validation base in order to measure the time consumed for the validation of their composition specifications (*i.e.*, unreachable state and transition search and deadlock trace extraction). As it can be observed in Table 1[3], the time elapsed for the validation of the examples experiences a linear growth with the number of traces generated, therefore scalability of the tool and the algorithms is satisfactory. Indeed, in the worst case, the validation of our biggest example using 20000 traces takes about 2 minutes, which is a reasonable amount of time.

Name	Size					Execution Time (s) vs. Number Traces generated			
	Number of Comps.	VLTS		Comps.					
		States	Tran	States	Tran	5000	10000	15000	20000
rate-finder-v1	3	5	12	21	19	4.8	9.9	14.8	19.4
broadcast-v12	3	14	21	186	555	26.3	53.5	79.9	106.2
mcomp-v1	6	23	37	393	1073	33.3	66.6	99.8	132.9

Table 1: Time elapsed for the validation of 3 examples using Clint

In this article, we have illustrated our approach on a rate finder service. Some other case studies, namely a travel agency and a push-out advertisement service, are available on the Clint webpage [CLINT 2007].

7 Run-time Adaptation in Practice using AOP

Aspect-Oriented Programming (AOP) is based on the idea that systems are better programmed by separately specifying the different concerns (areas of interest) of a system in *aspects* and a description of their relations, relying on mechanisms to *weave* or compose them into a working system. This weaving process can be performed at different stages of the development, ranging from compile-time to run-time (dynamic weaving) [Popovici et al. 2003]. The dynamic approach (Dynamic AOP or d-AOP) implies that the virtual machine or interpreter running

[3] The experiments were conducted on an Intel Centrino Duo T2300 running at 1.66GHz with 2GB of RAM installed.

the code must be aware of aspects and control the weaving process. This represents a remarkable advantage over static AOP approaches, considering that aspects can be applied and removed at run-time, modifying application behaviour during the execution of the system in a transparent way.

In addition, with conventional programming techniques, programmers have to explicitly call other components' methods in order to access their functionality, whereas the AOP approach offers implicit invocation mechanisms for invoking behaviour in code whose writers were unaware of the additional concerns (*obliviousness*). This implicit invocation is achieved by means of *join points.* These are regions in the dynamic control flow of an application (method calls or executions, exception handling, field setting, etc.) which can be picked up or intercepted by an AOP program by using *pointcuts* (expressions which allow the quantification of join points) to match on them. Once a join point has been matched, the AOP program can run the code corresponding to the new behaviour (*advices*) typically *before, after, instead* of, or *around* (before and after) the matched join point. Since join points are dynamic, it is possible to access run-time information such as the caller or callee of a method from a join point.

In order to implement our approach in full, we aim at modifying component communication through d-AOP by wrapping components with aspects able to capture all incoming/outgoing messages by means of pointcuts, and modify or substitute them conveniently through the application of advices. Specifically, Dynamic AOP enables us to tailor the adaptation engine with aspects able to: (i) intercept communication (*i.e.,* service invocations) between components; (ii) apply the algorithms described in this proposal in order to make the right message substitutions; (iii) forward the substituted messages to their recipients transparently.

First of all, as a prerequisite for run-time adaptation, we must incorporate our protocol par-LTS decriptions as metadata in components. Java annotations are readable at run-time through reflection. Specifically, custom-defined, multi-value type Java annotations, which have multiple data members, permit the inclusion of protocol information in components [Cámara et al. 2007a].

As regards the implementation of the adaptation engine, we have chosen PROSE [4] (PROgrammable extenSions of sErvices), which enables the modification of Java programs at run-time by applying and removing aspects at any time during execution. This is particularly interesting since our adaptation engine can take advantage of information which is only available at run-time. Aspects in PROSE extend the `DefaultAspect` base class, and may contain one or more crosscut objects. A crosscut object defines an advice method, and a pointcut method which defines a set of join-points where the advice should be executed.

Our adaptation engine needs to keep track of the state of the different com-

[4] http://prose.ethz.ch/

ponents during the interaction. In order to achieve this, we can implement a *tracking aspect* in the engine using a method exit advice. In such a way, each time a method is intercepted, the state of the component is updated in the adaptation engine just after the execution of the method.

In order to replace the communication protocol of the different components, the `MethodRedefineCut` class, similar to the `around` advice construct in AspectJ [Kiczales et al. 2001], permits the redefinition of a particular method. Using method redefinitions enables us to define an *interception aspect* in order to intercept and substitute method invocations with the ones specified by the algorithms.

Finally, the main body of the adaptation engine consists of a set of regular (*i.e.,* non aspectual) Java classes implementing the different algorithms described in this proposal. These classes are aided by the aforementioned aspects which provide the means to supply information about the state of components and the messages which are being exchanged among them. While receptions in the adaptation engine are delegated to the *interception aspect*, emissions can be performed making use of the method invocation facility included in the Java Reflection API.

8 Related Work

In this section, we will compare our approach with related works in software adaptation, especially those which enable run-time adaptation, and verification of systems under adaptation.

Software adaptation is a promising topic in software composition. Indeed, composition assumes that components will successfully interact when combined, whereas most of the components reused out of their original context cannot be integrated *as is*, requiring some degree of adaptation. Many proposals [Schmidt and Reussner 2002, Inverardi and Tivoli 2003a, Bracciali et al. 2005, Brogi et al. 2006b, Canal et al. 2006b] in this area focus on the behavioural interoperability level, and advocate abstract notations (*e.g.*, correspondences between messages, vector regular expressions, or LTL formulae) and state-of-art algorithms to derive adaptor protocols. As an example, an interesting proposal in this field is that of Inverardi and Tivoli (IT) [Inverardi and Tivoli 2003a]. Certain aspects of their work go beyond [Bracciali et al. 2005, Brogi et al. 2006b] by addressing the enforcement of some behavioural properties (namely liveness and safety properties expressed as specific processes) out of a set of already implemented behaviours. Starting from the specification with Message Sequence Charts of the components to be assembled and of the properties that the resulting system should satisfy, they automatically derive the adaptor glue code for the set of components in order to obtain a property-compliant system. In order to do that, they follow the so-called restrictive approach[5]. The IT proposal was extended in [Inverardi and Tivoli 2003b] with the use of temporal logic; coordination policies are expressed as LTL properties, and then translated into Büchi automata. Recent outcomes of this research line take into account time and other QoS issues [Tivoli et al. 2007].

To reduce the complexity involved in the generation of full adaptors, recent works aim at distributing the adaptation process [Autili et al. 2006] or at building adaptors incrementally [Poizat and Salaün 2007]. For instance, in [Poizat and Salaün 2007] the authors present a description of open component systems. Thus, software components distinguish in their description internal and external bindings, the latter ones being used for further connections with components or services to be added in the future. Moreover, they propose an incremental process for the integration and adaptation of open software components, enabling the construction of systems step-by-step (by adding or removing components), and to reconfigure them if necessary. Both approaches [Autili et al. 2006, Poizat and Salaün 2007] rely on algorithms which generate global adaptors. Hence, although this makes the complexity of the adaptor computation exponential, this boundary is purely theoretical and seldom reached in practice. Compared to these proposals, we also base our adaptation proposal on an expressive yet simple notation (vector LTS) and efficient algorithms, but we completely avoid the full generation of the static adaptor by applying the composition specification at run-time, adapting interaction whenever required.

Other works have promoted the use of models inspired by the π-calculus as behavioural descriptions of components, such as [Brogi et al. 2002, Bracciali et al. 2005]. In these papers, the authors use a π-calculus formalism although in a restricted way, keeping only value passing (hence discarding name passing). In our proposal, we chose to avoid system infiniteness issues coming from the use of the spawn operator which also has to be tackled in [Bracciali et al. 2005], and to deal with name creation and passing. This enables us to deal with run-time adaptation, and to implement prototyping tools (a lack of the work described in [Bracciali et al. 2005]).

As regards adaptation at run-time, most works deal with static architectures, reducing software systems to fixed structures of processes or components known at design time, whose interactions are described by using finite-state grammars [Yellin and Strom 1997], process algebras such as CCS [Inverardi and Tivoli 2003b], or non-recursive interaction pat-

[5] *Restrictive* approaches simply try to solve the problem by cutting off the behaviour that may lead to mismatch, thus restricting the functionality of the involved components. On the contrary, *generative* approaches try to accommodate the protocols without restricting the behaviour of the components, by generating adaptors that act as mediators, remembering and reordering events and data when necessary.

terns [Bracciali et al. 2005], to mention a few examples. In contrast, our approach adresses systems where the structure of the system is not fixed, since name passing allows the use of channels created at run-time, facilitating issues such as (private) communication between components which were not initially acquainted with each other.

Although most of the approaches adapting interacting components ensure correctness by construction, they need assessment techniques. Indeed, these adaptation approaches rely on an abstract description (a mapping) of how mismatches can be solved, and this mapping may be error-prone because it has to be designed by a human architect. In [Poizat and Salaün 2007], the authors focus on an incremental building of a system, and advocate for the use of alphabet comparison to compare both systems before and after the addition or suppression of a component. Furthermore, one may use model-checking techniques to check that the system behaves as required at each stage of the system's construction. Here, we want to avoid the generation of the full adaptor. Therefore, we prefer to generate traces using our prototype tool, and reason on the set of generated traces. Our verification techniques are completely automated (not the case in [Poizat and Salaün 2007] which requires to write out temporal formulas), and supported by graphical visualisation which makes easier the designer's understanding, helping to debug the composition specification.

To sum up, our proposal for run-time adaptation is innovative since it jointly gathers (i) efficient adaptation means, based on a simple yet expressive notation for the composition, as well as efficient algorithms to solve mismatch situations; (ii) a composition technique able to deal with systems where new channels are created and used at run-time; (iii) application of a composition specification at run-time, which avoids the costly generation of adaptors. We have also validated our techniques for composition and run-time adaptation through a prototype tool (a lack of most of the works mentioned above). Finally, we have shown how our proposal can be implemented in practice using Dynamic-AOP platforms.

9 Concluding Remarks

In this article, we have tackled compositional issues raised when building systems by reusing of existing components. Indeed, most components cannot be directly reused because of mismatches which have to be corrected. In addition, we have focused on systems where protocols are not entirely known at design-time due to the possible dynamic creation of channels. This is a realistic assumption that most works dedicated to adaptation discard. We have proposed an approach where our engine relies on a composition specification, applies it at run-time adapting interaction whenever required, and is able to handle channel instances created dynamically. We have implemented a prototype tool to try out our composition engine, and we have applied it on a large number of case studies. Last, we have showed how our formal proposal can be put in practice on real software components by using Dynamic AOP.

As regards future work, we will focus on two main perspectives. The first one aims at extending our Clint prototype to support the incremental construction of the composition specification. Indeed, the writing of this specification by a designer may be pretty hard and error-prone. On the other hand, approaches dedicated to the automatic generation of compositions are not mature enough. We are convinced that an assisted approach is a very good trade-off between complete automation and manual writing of the composition specification. Clint will be extended to accept a partial specification of the composition, point out composition issues (such as deadlocks in components), and propose possible solutions or further message correspondences to complete this specification. Our second perspective will focus on the implementation of our whole proposal in a middleware using Dynamic AOP following the ideas presented in Section 7.

Acknowledgements

The authors would like to thank Ernesto Pimentel and Pascal Poizat for interesting discussions and comments on this work. They are also grateful to the anonymous referees who helped to improve the contents and quality of this article.

References

[Andrews et al. 2005] T. Andrews et al. *Business Process Execution Language for Web Services (WSBPEL).* BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems, Feb. 2005.

[Arbab et al. 2002] F. Arbab., F. S. de Boer, M. M. Bonsangue, and J. V. Guillen Scholten. A Channel-based Coordination Model for Components. In *Proc. of FOCLASA'02*, volume 68(3) of *ENTCS*, 2002.

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

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

[Alfaro and Henzinger 2001] L. Alfaro and T.A. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.

[Brogi et al. 2002] A. Bracciali, A. Brogi, and C. Canal. Dynamically Adapting the Behaviour of Software Components. In *Proc. of COORDINATION'02*, volume 2315 of *LNCS*. Springer, 2002.

[Bracciali et al. 2005] A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1), 2005.

[Brogi et al. 2006a] A. Brogi, S. Corfini, J. F. Aldana, and I. N. Delgado. Automated Discovery of Compositions of Services Described with Separate Ontologies. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*. Springer, 2006.

[Brogi et al. 2006b] A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Science of Computer Programming*, 2006. To appear.

[Ben Mokhtar et al. 2005] 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, 2005.

[Cámara et al. 2007a] J. Cámara, C. Canal, J. Cubo, and J. M. Murillo. An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. In *Proc. of WCAT'06*, volume 189 of *ENTCS*. Elsevier, 2007.

[Cámara et al. 2007b] J. Cámara, G. Salaün, and C. Canal. Run-time Composition and Adaptation of Mismatching Behavioural Transactions. In *Proc. of SEFM'07*, IEEE, 2007.

[CLINT 2007] Clint v01/September 2007 distribution (LGPL licence). http://serv-linux.lcc.uma.es/clint/, 2007.

[Canal et al. 2006a] C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation: an Introduction. *L'Objet*, 12(1), 2006.

[Canal et al. 2006b] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*. Springer, 2006.

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

[Ingolfsdottir and Lin 2001] A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*. Handbook of Process Algebra. Elsevier, 2001.

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

[Inverardi and Tivoli 2003b] P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. In *Formal Methods for Software Architectures*, volume 2804 of *LNCS*. Springer, 2003.

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, volume 2072 of *LNCS*, 2001.

[Knuth 1969] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[Lynch and Tuttle 1989] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Milner 1999] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[Magee et al. 1999] J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour Analysis of Software Architectures*, pages 35–49. Kluwer Academic Publishers, 1999.

[Montanari and Pistore 1997] U. Montanari and M. Pistore. An Introduction to History Dependent Automata. In *Proc. of HOOTS II*, volume 10 of *ENTCS*. Elsevier, 1997.

[Popovici et al. 2003] A. Popovici, A. Frei, and G. Alonso. A Proactive Middleware Platform for Mobile Computing. In *Proc. of Middleware'03*, volume 2672 of *LNCS*. Springer, 2003.

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

[Ramsokul and Sowmya 2006] P. Ramsokul and A. Sowmya. ASEHA: A Framework for Modelling and Verification of Web Services Protocols. In *Proc. of SEFM'06*. IEEE Computer Society, 2006.

[Schmidt and Reussner 2002] H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS'02*. Kluwer, 2002.

[Tivoli et al. 2007] M. Tivoli, P. Fradet, A. Girault, and G. Goessler. Adaptor Synthesis for Real-Rime Components. In *Proc. of TACAS'07*, volume 4424 of *LNCS*. Springer, 2007.

[UDDI 2000] UDDI Consortium. *UDDI Technical White Paper*, September 2000.

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