Run-time Composition and Adaptation of Mismatching Behavioural Transactions

Javier Cámara, Gwen Salaün, and Carlos Canal
Department of Computer Science, University of Málaga, Spain
Emails: {jcamara,salaun,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 mismatching behavioural interfaces. Here, we particularly focus on systems for which the number of transactions is unbounded, and unknown in advance. This is typical in pervasive systems where a new client may show up at any moment to request or access a specific service. Hence, we advocate for the use of the π-calculus to specify component interfaces. The π-calculus is particularly suitable for creating new component instances and channels dynamically. The unbounded number of transactions and the use of the π-calculus obliges to apply the composition at run-time. In this paper, we propose a run-time composition engine that solves existing mismatches.

Keywords: component-based development, behavioural interfaces, mismatch, software adaptation, run-time composition, π-calculus.

1 Introduction

Software components (or services) are seldom reusable as is because of mismatches that may appear at different levels [4]: signature, behaviour, quality of service and semantics. 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 [17, 4]. To this purpose, model-based adaptation techniques are based on behavioural component descriptions and a composition expression, i.e., an abstract description of the interactions and adaptations to be applied to make all the components work together correctly.

Most of existing adaptation approaches [17, 16, 8, 3, 5] build component adaptors for the whole system, which is a costly process. The adaptation process relies on the specific number of components and transactions involved in the system, and the adaptor has to be recomputed each time a new entity is taken into account. These static approaches are therefore not suitable in pervasive systems where the number of transactions is not known in advance: a new client may show up at any moment to request or access a specific service. We propose to use the π-calculus [11] to specify component interfaces within this kind of system. The π-calculus is particularly appropriate for dealing with the addition of components involved in a new transaction at run-time thanks to operators which allow the creation of component instances, the creation of new channel names and their passing through channels.

The unbounded number of transactions and the use of the π-calculus oblige to apply the composition and the resolution of potential mismatch situations (as defined in the composition expression) at run-time. This also avoids the costly generation of a new adaptor every time the system has to deal with a new transaction. However, while composing the components dynamically, we should avoid the execution of undesired behaviours that may lead to deadlock situations. Such deadlocks are statically removed in the approaches generating full descriptions of the adaptors such as those mentioned above. As a result, we propose a composition engine which is capable of composing a set of components at run-time and also detecting branches unable to terminate, thus avoiding deadlocking executions.

This paper is structured as follows: Section 2 introduces our component model, and the case study we use throughout this paper. 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 composition engine which applies at run-time a composition expression. Section 5 compares our approach with related works. Section 6 ends the paper with some concluding remarks.
2 Component Interfaces

In this section, we introduce successively our component model based on the $\pi$-calculus, a running example, and a process to derive Labelled Transition Systems (LTSs) from the components, which are used as input for the algorithms described in Section 4.

2.1 Behavioural Interfaces

Component interfaces are given using a signature and a behavioural interface. A signature is a set of provided and required operations. An operation profile is the name of an operation, together with its argument types, its return type and the exceptions it raises. We extend signatures with behavioural interfaces specified using the $\pi$-calculus. We have chosen this language because it is simple yet expressive enough to this purpose. Moreover, creation and use of new channels and components is possible with the $\pi$-calculus. This is not the case in other process calculi such as CCS, CSP or LOTOS. Messages (or events) involved in process algebraic descriptions of the component correspond to the operations used in its signature. Here, we focus on the behavioural interoperability level and therefore we do not deal with operation arguments, return values or exceptions.

Definition 1 ($\pi$-calculus) A process $P$ in $\pi$-calculus is defined using the following constructs:

$$P ::= 0 \mid \alpha(x).P \mid \alpha(y).P \mid P_1+P_2 \mid !A \mid (\nu x)P \mid P||Q$$

The specific process 0 stands for termination. Each process can be prefixed by an emission $\alpha(y)$, a reception $\alpha(x)$, or composed with other processes using the parallel $\mid$ or the choice $+$ operator. The spawn operator $!A$ creates a new instance of process $A$. Finally, $(\nu x)P$ represents the creation of a new name $x$ whose scope is $P$.

The $\pi$-calculus only defines termination states as deadlocks using 0, and does not allow to define final states which are not always terminating (e.g., in case of a process which may terminate after several recursive calls). Accordingly, we extend this notation to tag processes with explicit final state attributes using $[f] \ (0 [f] = [f])$.

In addition, we restrict the $\pi$-calculus semantics in two ways. First, the spawn $!A$ operator may lead to infinite behaviours. This work however, focuses on finite transactional descriptions computed from component interfaces specified using the $\pi$-calculus. Hence, potential infinite behaviours derived from the application of the spawn operator are cut away. Second, the parallel composition rule expressing communication is not used since we propose our own communication model between components in Section 3.

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, or either a message in one component has no counterpart or matches with several messages (e.g., in case of broadcast communication). We will illustrate specific cases of mismatch in the case study we use as running example in Section 2.2. More formally, cases of mismatch lead the whole system to deadlock states. A deadlock state is a state which is not final and in which the process cannot evolve. A system deadlocks when all its constituent components are blocked because at least one of them is in a deadlock state.

2.2 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.

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 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. This interface can be described in $\pi$-calculus as follows:

Client [f] = requestRate!,lowestRate?.
(\text{connect!}.\text{session}?(\text{sid}),\text{ConnClient} + \text{end}! [f] )

ConnClient [f] = sid!(\text{serv}).\text{CCAux}
CCAux = sid?(\text{access}).\text{CCAux} + sid!(\text{quit}! [f])

The Service interface receives requests to find the provider with the lowest rates. Once the service receives a request, it creates a new thread (ConnService) to serve this request on a new private communication channel $m$, which is returned to the client (sendServ!(m)). This newly created thread returns the requested information first (m!(sendFare)), and then 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:

Service [f] = request?!,!(Service||\nu(m).sendServ!(m).ConnService)
ConnService = m!(sendFare).
(m?(connect).\nu(conn).m!(conn) [f] + m?(end) [f])
The Provider interface receives incoming client requests on a specific communication channel conn(open?(conn)), and then it creates a new thread ConnProvider to interact with the connected client. ConnProvider receives incoming service requests and then sends back service access to the client:

\[
\begin{align*}
\text{Provider}[\ell] &= \text{open}?(\text{conn}).(!\text{Provider}||\text{ConnProvider}) \\
\text{ConnProvider}[\ell] &= \text{conn}!(\text{serv}).\text{conn}!(\text{access}).\text{ConnProvider}
\end{align*}
\]

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 ml!(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 (ml!(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 (ml!(sendFare)), resulting in a mismatch.

2.3 Generating LTS from \(\pi\)-calculus

In the remainder, while presenting our run-time composition approach, we use LTSs as behavioural interfaces. Although the \(\pi\)-calculus provides a more expressive notation which simplifies their description, the LTS notation is more appropriate for developing composition algorithms since they rely on the traversal of the different states of the components. Indeed, process algebraic descriptions do not provide an explicit description of this set of states and their relations, whereas the transition function of LTS descriptions makes the access to this information straightforward.

Definition 2 (Component LTS) A Component LTS is a tuple \((A, S, I, F, T)\) where: \(A\) is an alphabet which corresponds to the set of events on the component’s interface, \(S\) 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.

These component LTSs can be automatically generated from \(\pi\)-calculus processes using the operational rules of the process algebra (see [11] for the Structural Operational Semantics of the \(\pi\)-calculus). However, this is achieved using a different interpretation for some SOS rules compared to the original ones. First, our LTS component model encodes parallel process composition \(||\) as a concurrent behaviour without communication. This is achieved by introducing fork states. These states are identified by two 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. Second, potential infinite behaviours derived from the application of the spawn operator are cut away: spawning is equivalent in our model to a final state.

An LTS \((A, S, I, F, T)\) from a process \(P\) described in the \(\pi\)-calculus is incrementally built by: (i) first, assigning the initial state \(s_0\) of the LTS to the implicit initial state of \(P\), (ii) computing the set of possible transitions for the process applying the SOS rules, (iii) deriving states, alphabet, and final states from the set of transitions previously computed.

- \(I = s_0\),
- \(T = \{(s, \alpha, s') \mid (s, P) \xrightarrow{\alpha} (s', P') \wedge \neg\text{fork}(P')\} \cup \{(s, \alpha, (s'_1, s'_2)) \mid (s, P) \xrightarrow{\alpha} (s'_1, P') \wedge \text{fork}(P')\}\)

where \((s, P) \xrightarrow{\alpha} (s', P')\) are inferred from the SOS rules and \(s, s'\) are identifiers that characterise the state in which \(P\) is at a specific point of its execution.

- \(S = \{I\} \cup \{s' \mid (s, \alpha, s') \in T\} \cup \{(s'_1, s'_2), s'_1, s'_2\} \mid (s, \alpha, (s'_1, s'_2)) \in T\}\)
- \(A = \{\alpha \mid (s, \alpha, s') \in T\}\),
- states in \(F\) correspond to: (i) states tagged with \([\ell]\), or (ii) states of the process which correspond to the application of the spawn operator.

The fork function for a given process \(P\) is defined as:

\[
\text{fork}(P) = \begin{cases} 
\text{true} & \text{if } P = P_1 || Q \\
\text{false} & \text{otherwise}
\end{cases}
\]

Notice that we use LTS only as a structure, but with our own interpretation, since fork states correspond to concurrent running branches. Hence, the LTS can have several
active states at a time. In order to interpret the LTS, we define the run function, which takes as input the current set of active states of the LTS (initially \( I \)), an event \( \ell \), and the transition function \( T \), and returns the new set of active states after having triggered one applicable transition:

\[
\text{run}(\{s_0, \ldots, s_n\}, \ell, T) = \begin{cases} 
\{s_0, \ldots, s_j - 1, s'_j, s_{j+1}, \ldots, s_n\} & \text{if } s_j \in \{s_0, \ldots, s_n\} \land (s_j, l, s'_j) \in T \\
\{s_0, \ldots, s_j - 1, s'_j, s_{j+1}, \ldots, s_n\} & \text{if } s_j \in \{s_0, \ldots, s_n\} \land (s'_j, l, s_j) \in T \\
\bot & \text{otherwise (no transition applicable)} 
\end{cases}
\]

**Example.** To illustrate the translation of a \( \pi \)-calculus process into an LTS, consider the specification of the Service component in our case study where, once a client makes a request, the service creates a new thread which handles the rest of the interaction with that particular client. Once we designate \( I = s_0 \), we begin by defining the set of possible transitions between states in the process:

\[
T = \{(s_0, \text{request}!), (s_{1l}, s_{1l}), (s_{1l}, \text{sendServ}!(m), s_2), (s_2, m!(\text{sendFare}), s_3), (s_3, m?(\text{end}), s_4), (s_3, m?(\text{connect}), s_5), (s_5, m!(\text{conn}), s_6)\}
\]

Subsequently, the set of states and the alphabet are 
\[
S = \{s_0, s_{1l}, s_{1r}, s_1, s_2, s_3, s_4, s_5, s_6\}
\]

and 
\[
A = \{\text{request}?, \text{sendServ}!(m), m!(\text{sendFare}), m?(\text{end}), m?(\text{connect}), m!(\text{conn})\}.
\]

The set of final states is 
\[
F = \{s_0, s_4, s_{1l}, s_1, s_2, s_3, s_4, s_5, s_6\}
\]

and \( s_{1r} \) is explicitly tagged with \([\ell] \), and \( s_{1r} \) is derived from the application of the spawn operator. The resulting LTS is depicted in Figure 1, where initial and final states are respectively noted using bullet arrows and darkened states.

### 3 Composition Language

In this section, we present our composition language that makes communication between components explicit, and specifies how to work out mismatch situations. This is achieved by means of synchronisation 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.

**Definition 3 (Synchronisation Vector)** A vector for a set of components \( C_i, i \in \{1, \ldots, n\} \), specified using the \( \pi \)-calculus, is a tuple \( \langle i_1, \ldots, i_n \rangle \) with \( i_l \in A_i \cup \{\varepsilon\} \), where \( A_i \) are the alphabets of components \( C_i \) and \( \varepsilon \) 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., \( \langle c_1 : \text{comm}!, c_2 : \varepsilon, c_3 : \text{comm}? \rangle \).

We use as abstract notation to describe 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 precise vectors in a specific order. If only correspondences without a specific order are necessary between components, the vector LTS contains a single state with all vector transitions looping on it.

**Definition 4 (Composition Language)** A composition language for a set of components \( C_i, i \in \{1, \ldots, n\} \), is a couple \((V, C_{cc})\) where \( V \) is a set of vectors for components \( C_i \), and \( C_{cc} \) 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 writing of the composition expression. Let us consider two components, \( C_1 \) and \( C_2 \), exchanging login and request information, with messages that have to be reordered to make the communication possible: \( C_1 = \text{log!}\cdot\text{req}!(\ell), C_2 = \text{query}?.\text{id}?[\ell] \). Our approach can reorder messages using the following vectors: \( \langle c_1 : \text{log}!, c_2 : \varepsilon \rangle, \langle c_1 : \text{req}!, c_2 : \text{query}?! \rangle \), and \( \langle c_1 : \varepsilon, c_2 : \text{id}?! \rangle \), in which we specify that the interaction on \( \text{log} \) is desynchronised, temporarily memorised until its use for effective interaction on \( \text{id} \). This leads to the following execution trace: \( c_1 : \text{log}!, c_1 : \text{req}!, c_2 : \text{query}?, c_2 : \text{id}?! \).

**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:

\[
\begin{align*}
\nu_{\text{req}} &= \langle c : \text{requestRate}!, s : \text{request}?!, p : \varepsilon \rangle \\
\nu_{\text{serv}} &= \langle c : \varepsilon, s : \text{sendServ}!(m), p : \varepsilon \rangle \\
\nu_{\text{far}} &= \langle c : \text{lowestRate}?, s : m!(\text{sendFare}), p : \varepsilon \rangle
\end{align*}
\]
4 Run-time Composition

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

Our proposal is able to deal with an unbounded number of transactions. It means that any time a new transaction is run, a new composition engine instance has to handle all the component instances belonging to this transaction. Figure 3 gives an overview of the LTS interfaces and the composition engine used in our running example for each new client session. 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 expression at run-time resulting in the dynamic execution of the involved components.

4.1 Existence of Global Final States

Although the composition expression aims at solving mismatch cases that correspond to deadlocks, its application can still lead to remaining deadlocks. Indeed, the composition expression is an abstract description of how components work together, and does not take into account all the possible execution scenarios. Removing these remaining spurious interactions is required to let the system reach a final state. Since the composition expression is applied at run-time, it is not possible to apply beforehand the removal of deadlocks as it is the case in static coordination and adaptation approaches [8, 5]. Therefore, before applying a vector which belongs to the composition expression, we check that next to this vector, there exists at least one global final state for the whole system. Thus, our composition engine will prevent the system to end 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 LTSs $C_i$, the composition expression $C_{cc}$, a vector $v$, and the current state of the system (i.e., the current states of the components as well as the current state $s_{cc}$ of the composition expression). 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 expression 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.

In the inside while loop, we introduce an abstract fixpoint condition ($\text{fixpt}$) that enables the application of a vector even if it leads to an already traversed state of the composition expression. This is necessary since in some cases, several executions of a cycle in the vector LTS may finally lead to a termination state. This fixpoint condition is bounded to avoid non-terminating executions.

**Algorithm 1**

\textbf{Algorithm 1 exist\_final}

\textbf{inputs} states, compo. $C_i \in \{C_1, \ldots, C_n\} = (A_i, S_i, I_i, F_i, T_i)$, $s_{cc}, C_{cc} = (A_{cc}, S_{cc}, I_{cc}, F_{cc}, T_{cc})$, and a vector $v$

\textbf{outputs} a boolean

1. visited := {}  \\
2. current := $s_{cc}$  \\
3. path := []  \\
4. cloop := true  \\
5. while final$(s_{cc}, F_{cc}, C_{cc})$ and cloop do
6. states := next\_states$(v, states, T_{cc})$
7. current := next$(current, v, T_{cc})$
8. path := append(path, (current, states))
9. visited := visited $\cup$ {current}
10. while final$(states, F_{cc}, C_{cc})$ and $v = v_\perp$ and path $\neq$ [] do
11. v := applicable$(states, C_i, current, C_{cc})$
12. cond := visited $\cup$ next$(current, v, T_{cc})$ $\cup$ fixpt
13. if $v = v_\perp$ or cond then
14. current := left$(last\_path)$
15. states := right$(last\_path)$
16. path := remove$\_last\_path$
17. end if
18. end while
19. cloop := ($v \neq v_\perp$)
20. end while
21. return final$(states, F_{cc}, C_{cc})$

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

\[
\text{final}(states, F_{cc}, C_{cc}) = f(states[1], F_1) \land \ldots \land f(states[n], F_n) \land s_{cc} \in F_{cc} \\
f([s_1, \ldots, s_m], F) = s_1 \in F \land \ldots \land s_m \in F
\]

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

**Example.** We focus on a piece of our running example to illustrate how the final state detection is used to avoid a deadlock situation. We recall first the part of the client specification which requests a service, can access it as many times as needed, and at some point quits:

\[
\text{ConnClient} \{\text{f} \} = \text{sid!}(\text{serv}).\text{CCAux} \\
\text{CCAux} = \text{sid?!}(\text{access}).\text{CCAux} + \text{sid!}(\text{quit}) \{\text{f}\}
\]

On the other hand, the provider waits for a request and provides a service:
4.2 Run-time Composition Engine

This section presents our run-time composition engine which deals with the execution of one transaction. A new execution of the algorithm we present below is fired for each new transaction (this will correspond to a new client requesting the rate finder service in our example). The composition is applied at run-time since new channel names are created at run-time and cannot be known beforehand. The right substitutions of event names are dynamically performed by the algorithm, and 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 LTSs with respect to a given composition expression. We propose a composition which respects the sequential interactions described within vectors of the composition expression, that is events belonging to two different vector transitions of the composition expression are never interleaved. Such an interleaving may happen in some cases where 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 composition 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 (scce) as well as of the current states of the components (states). The algorithm ends when the global system has reached a final state. Note that 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 ≠ v⊥), otherwise the composition is not launched. Notice the abstract loop condition introduced in order to control iterative behaviours within the current transaction. 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.

\[
\text{ConnProvider} \{f\} = \text{conn}?(\text{serv}).\text{conn}!(\text{access}).\text{ConnProvider}
\]

Vectors v_{serv}, v_{acc}, v_{serv}, v_{quit} in Figure 2 are those dedicated to this part of the composition. We show in Figure 4 the transition system summarising all the possible applications of these vectors with respect to a given component (ConnClient and ConnProvider interfaces. After the application of v_{serv} (state s in Fig. 4), ConnClient 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 ConnProvider 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, and therefore enforces the execution of v_{acc}.

![Figure 4. Deadlocking application of vectors](image)

**Algorithm 2 run-time composition**

composes at run-time a set of components wrt. a composition expression

\[
\text{inputs component } C_i \in \{1, \ldots, n\} = (A_i, S_i, I_i, F_i, T_i), \text{ compo. } C_{ce} = (A_{ce}, S_{ce}, I_{ce}, F_{ce}, T_{ce})
\]

1: states := \{[I_1], \ldots, [I_n]\} // current states in C_i
2: E_i := \emptyset, E_{ce} := \emptyset // substitution environments
3: s_{ce} := I_{ce} // current state in the compo. exp.
4: v := select_vector(states, C_i, s_{ce}, C_{ce}, E_i)
5: while v ≠ v⊥ \& \ (¬final(states, F_i, s_{ce}, C_{ce}) ∨ loop) do
6: \quad \text{Rec} := \text{emissions}(v, E_i)
7: \quad x := \text{receptions}(v, E_i)
8: \quad \text{repeat } \text{Var}()
9: \quad \text{repeat } // effective receptions
10: \quad r?(x) | r ∈ \text{Rec}, j ∈ \{1, \ldots, n\}, s_j = \text{states}[j], l ∈ A_j, \text{run}(s_j, l, T_j) = s_j′, v = \text{obs}(l, E_j)
11: \quad \text{Rec} := \text{Rec} \setminus \{r\}
12: \quad \text{states}[j] := s_j′
13: \quad \text{until } \text{Rec} = \emptyset
14: \quad \text{repeat } // effective emissions
15: \quad e!(x) | e ∈ \text{Em}, j ∈ \{1, \ldots, n\}, s_j = \text{states}[j], l ∈ A_j, \text{run}(s_j, l, T_j) = s_j′, v = \text{obs}(l, E_j)
16: \quad \text{Em} := \text{Em} \setminus \{e\}
17: \quad \text{states}[j] := s_j′
18: \quad E_j := E_j ∪ \{(\text{par}(l) ← \$x)\} // $x$ is the value of var. x
19: \quad \text{until } \text{Em} = \emptyset
20: \quad s_{ce} := \text{next}(s_{ce}, v, E_{ce})
21: \quad v := select_vector(states, C_i, s_{ce}, C_{ce}, E)
22: \quad \end while

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:

\[
\text{select_vector}(\text{states}, C_i, s_{ce}, C_{ce}, E) = \begin{cases} v & \text{if } v = \text{applicable}(\text{states}, C_i, s_{ce}, C_{ce}, E_i) \\ v \neq v⊥, \text{exist_final}(\text{states}, C_i, s_{ce}, C_{ce}, v) & \text{otherwise (no vector applicable)} \end{cases}
\]
We define *emissions* and *receptions* which return the set of emissions and receptions respectively, of any given synchronisation vector. These functions also substitute formal names used in the vectors by the actual ones appearing in the environment $E$:

\[
\text{emissions}(l_1, \ldots, l_n, E_i) = \text{em}(l_1, E_i) \cup \ldots \cup \text{em}(l_n, E_n)
\]

\[
\text{em}(l, E) = \begin{cases} \{\text{sub}_n(e, E)\} & \text{if } l = e!x \\ \emptyset & \text{if } l = r?x \lor l = \varepsilon \end{cases}
\]

\[
\text{receptions}(l_1, \ldots, l_n, E_i) = \text{rec}(l_1, E_i) \cup \ldots \cup \text{rec}(l_n, E_n)
\]

\[
\text{rec}(l, E) = \begin{cases} \{\text{sub}_n(r, E)\} & \text{if } l = r?x \\ \emptyset & \text{if } l = e!(x) \lor l = \varepsilon \end{cases}
\]

where

\[
\text{sub}_n(e, E) = \begin{cases} E(e) & \text{if } e \in \text{dom}(E) \\ e & \text{otherwise} \end{cases}
\]

The *observational part* of an event $l$ is defined as:

\[
\text{obs}(e!x, E) = \text{sub}_n(e, E), \text{obs}(r?x, E) = \text{sub}_n(r, E)
\]

The \setminus operator denotes the set difference defined as: $E_1 \setminus E_2 = \{x \mid x \in E_1 \land x \notin E_2\}$. The abstract function $\text{generateVar}$ derives new variables when required.

The complexity of Algorithm 2 is polynomial – $O(|S_{cc}|^2)$. 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 $\text{exist\_final}$ is called, and in the worst case, the vector LTS is completely traversed.

**Proof (intuition).** We focus on the *correctness* of the algorithm. 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 expression, the process may yield a system in which no interactions are possible in case of having an incorrect or under-specified composition expression. 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 expression. 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 $\text{exist\_final}$ algorithm), and the algorithm will make all the components converge to this final state. The while loop may end when one global final state is reached.

**Example.** Figure 5 gives a possible execution scenario that is obtained running the composition 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 composition 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 interactions, the first one ($\text{requestRate}$) between the client and the engine, and another one ($\text{request}$) between the engine and the server.

During composition, 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 $v_{\text{sses}}$ in the mid-bottom of Figure 5), it can be observed that environments $E_c$ and $E_p$ are extended associating their formal names ($\text{sid}$ and $\text{conn}$, respectively) which appear in their interfaces and in the vector by the actual one which is received in $x_4$, namely $\text{conn}$. Environments are also necessary to make interactions between the engine and the components work correctly. For example, while applying vector $v_{\text{serv}}$, the formal name $\text{sid}$ in $\text{sid}(\text{serv})$ is replaced using the actual name $\text{conn}$. This correspondence is achieved using the couple ($\text{sid}$, $\text{conn}$) in $E_c$, assigned while executing $v_{\text{sses}}$.

In some cases, the engine receives a value, without forwarding it (see vector $v_{\text{sserv}}$ in the top part of Figure 5). Likewise, a vector may express a correspondence between a message without parameter, and a message with one (e.g., $v_{\text{conn}}$). In this case, the engine receives no value, and emits
Software adaptation is a promising topic in software composition. Indeed, composition assumes that the components will interact successfully when combined, whereas most of the components reused out of their original context cannot be integrated as is and need some adaptations. Many proposals [8, 3, 5] in this area focus on the behavioural interoperability level, and advocate abstract notations (correspondences between messages, regular expressions of vectors, or LTL formulae) and state-of-art algorithms to derive adaptor protocols. To reduce the complexity that the generation of full adaptors induces, recent works aim at distributing the adaptation process [1] or at building adaptors incrementally [14]. 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 thanks to an application of the composition at run-time.

Other works have already promoted the use of the π-calculus as behavioural descriptions of components, such as [2, 3]. In [2], the authors use the π-calculus although in a restricted way, forbidding looping behaviours, thus restricting the situations in which adaptation can be applied. In a subsequent paper, the same authors [3] deal with the full expressiveness of this notation, but in this case this prevents the implementation of tool support due to the induced infiniteness of systems. In our proposal, we have chosen a trade-off between these two attempts by restricting the ways the spawn operator can be used, but making possible to deal with name creation and passing. This allows to deal with finite transactional compositions possibly launched at runtime, and to implement prototyping tools.

As regards the unbounded number of transactions, most of the works focus on systems with a static architecture, reducing software systems to fixed structures of processes or components known at design time, whose interactions are described by using finite-state grammars [17], process algebras such as CCS [9], or non-recursive interaction patterns [3], to mention a few examples. Our approach addresses dynamic systems in which new components and transactions are handled at run-time.

Some recent proposals are based on run-time composition to tackle issues such as context-aware composition [10, 6], or dynamic (re)composition of (web) services and components [12, 13]. The interest in run-time composition is particularly vivid in the field of Web Services (WS), where several notations (e.g., WSBPEL and OWL-S) have been recently defined for describing composition and coordination of simple WS components in order to create more complex value-added services. In [13], the authors address the problem of automated composition of OWL-S process models using state-transition systems, and define on-the-fly compositions as those specifically created at runtime for satisfying a given request, but only offer to the client a simple (request/reply) interaction pattern. In [13], a new composition is completely generated for each new request whereas our goal is to avoid such a full generation of the composition beforehand. Moreover, adaption of service names and protocols are not addressed in these works, that deal more with coordination (referred to as orchestration in WS terminology) of composite services.

As far as the works dealing with recomposition and context-aware composition [10, 12, 6] are concerned, these proposals dynamically apply changes in the composition at runtime depending on some environment changes. However, these flexible compositions are defined at design-time.
and consequently in this case again the coordination of the involved entities follow a predefined interaction pattern.

To sum up, our proposal for software composition 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 take into account any number of transactions created at run-time (in contrast with previous proposals which deal with fixed system structures); (iii) run-time composition, which means that we avoid the costly generation of adaptors, since composition is applied as incoming transactions start within the system. Finally, we have validated our technique for run-time composition and adaptation implementing a prototype (a lack of most of the works mentioned above).

6 Concluding Remarks

In this paper we have presented an approach to the composition and adaptation of mismatching components in systems where the number of transactions is not known in advance. Our approach applies composition at run-time with respect to a composition specification, and enables the engine to handle component instances created dynamically. We have implemented a prototype in Python to try out our composition engine on a large number of examples.

Implementation issues. In order to implement our approach in full, we intend to use Dynamic Aspect-Oriented Programming (Dynamic AOP) [15]. Unlike in traditional platforms and languages, a particular system can be modified without altering its code in AOP by separately specifying modifications or aspects, and describing their relationship with the current system. Then the AOP environment composes or weaves both the original program and aspects into a coherent program. We are especially interested in the dynamic approach to AOP since aspects can be applied at run-time in a transparent way. Specifically, Dynamic AOP enables us to tailor the composition 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.

Future work. Our main goal is to implement the whole proposal in a middleware using Dynamic AOP. A second idea is to extend our run-time composition algorithm to allow the overlapping application of several vector transitions when possible. Indeed, our current algorithm applies the interactions appearing in one vector before executing a new one. Ultimately, we aim to ensure the correctness of the composition expression by applying verification techniques to the set of traces generated by our composition engine.

References