Behavioural Self-Adaptation of Services in Ubiquitous Computing Environments

Javier Cámara, Carlos Canal, Gwen Salaün
Department of Computer Science, University of Málaga, Spain
{jcamara,canal,salaun}@lcc.uma.es

Abstract

Self-adaptive software systems are those able to manage changing operating conditions dynamically and autonomously. Currently, most proposals in this field rely on an explicit representation of the constituent elements and goals of the system. This approach is suitable for systems where constituent elements are well known at design time. However, in systems where new elements may dynamically enter or leave the execution environment, it is not possible to obtain a predefined description of the system's architecture nor a complete specification of its goals. Paradigmatic examples of such systems can be found in ubiquitous computing, or dynamic web service discovery and composition, where new adaptability problems arise, such as the (dis)connection of unforeseen elements to an already running system, or ensuring properties of the composition among services, which cannot be addressed at static verification time since the state space of the system is not closed anymore. In this paper, we present our approach for the composition and resolution of interface mismatch among services in ubiquitous computing environments, dynamically reconfiguring the system as new services are integrated or disconnected.

1 Introduction

The proliferation of software services[1] and ubiquituous computing, as well as other factors, such as the increasing need of communicating systems spanning across organizational and physical boundaries, are shaping a world where the flow of information and connectivity between previously unacquainted systems and devices tends to be necessary in many situations. In particular, the need to discover and correctly access services as users move from one location to another and the conditions of the environment change, is a crucial requirement in the design and implementation of ubiquitous computing environments.

However, in most cases software services are not designed to interoperate with each other, since the different scenarios in which services may be reused cannot be envisioned *a priori* by their designers. This results in the appearance of mismatch situations among the public interfaces of services when we try to compose them. Specifically, four interoperability levels can be distinguished in Interface Description Languages (IDLs):

- *Signature*. Deals with the static aspects of service interoperability. At this level, IDLs (*e.g.,* public interfaces of Java classes, or WSDL descriptions in the case of Web Services) provide operation names, type of arguments and return values. Interoperability problems at this level are for instance operation renaming or parameter reordering between provided and required operations on the different interfaces.
- *Protocol* or *behaviour*. Specifies the order in which the operations available on an interface should be invoked (*i.e.,* the interactive behaviour that a service follows and expects from its environment). Problems at the protocol level involve bad ordering on the execution of operations because of differences in the behaviour of the services involved, which may lead to undersirable situations, such as *deadlocks* or infinite loops. Behavioural descriptions are always required for stateful services since, unlike in stateless services where an operation is available for invocation anytime, operation availability depends on the internal state of the service (described by its behavioural interface). Notorious examples of behavioural IDLs are Abstract BPEL, Windows Workflows (WF), or WS-CDL, used to describe Web Service orchestrations and choreographies.
- *Service*. Description of non-functional properties like temporal requirements, security, cost, etc. Quality of Service (QoS) descriptions and their related notations, such as the QoS Modeling Language (QLM), are usually highly customizable, and the possible specifications include mean values, standard deviations and a

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

set of quantiles characterizing the distribution of any self-defined quality metric.

- *Semantic*. This level concerns service functional specifications (i.e. what they actually do). Even if services present perfectly matching signatures, follow compatible protocols, and are also compatible at the service level, we must ensure that they are going to behave as expected. Hence, this level of description provides semantic information about services using ontology-based notations such as OWL-S (used in Web Services), which are interesting for service mining.

Software Adaptation [11, 37] is a hot topic in Software Engineering, since it is the only way to compose non-intrusively black-box services with mismatching interfaces by automatically generating mediating *adaptor* services able to solve interoperability problems at the aforementioned four levels.

This work addresses the problem of service adaptation in software systems where changes in the execution environment are directly subject to the availability of a particular service which may join (be discovered) or disappear from the context of the system in any given moment. Paradigmatic examples of such systems can be found in ubiquitous computing environments, or dynamic web service discovery and composition. Particularly, we focus on the protocol or behavioural level, currently acknowledged as one of the most relevant research directions in software adaptation [2, 3, 4, 6, 27, 37].

In order to illustrate our approach, we will use a running example which consists of a set of services described in the context of an airport: Let us suppose a traveller who walks into an airport with a handheld device (*e.g.,* a PDA) equipped with a client containing a task specification based on the different services which may be required while at the airport. These services may be accessed through a local wireless network. First, the user needs to contact his airline and check-in in order to obtain a seat on the flight. This is achieved by approaching a kiosk which provides a local check-in service available to the handheld device and prints a boarding card for the passenger. Next, the traveller may browse the duty-free shops located at the airport, and locally search for the different offers from any of them. The selected shop should be able to access the airport information system in order to check if a passenger has checked-in on a particular flight, and apply a tax exemption on the sale in that case. The payment will be completed by means of the credit and bank account information stored in the traveller's device, not requiring the use of a physical credit card. All these interactions must be accomplished without human intervention to solve interoperability problems.

This paper presents our approach for the composition and resolution of potential interoperability issues among services, as those presented above. Specifically, this work addresses two important problems:

Lack of a predefined architectural description. Although self-adaptive software systems are able to autonomously manage changing operating conditions and derive architectural descriptions from a set of system goals, most proposals currently rely on an explicit representation of the elements and goals of the system. These proposals are suitable for systems where constituent elements are well known at design time (*e.g.*, industrial control systems, robotics, unmanned vehicles, etc.), normally due to their close dependency on hardware parts. However, the constituent elements of a system in ubiquitous computing environments are known only at run-time, and the relations among them are volatile, changing constantly.

Verification of behavioural properties at run-time. Ensuring properties of the composition among services in ubiquitous environments is not possible at static verification time since the state space of the system is not closed.

In order to address these two problems, this paper contributes: **(i)** a new service interface model, which enables the derivation of an architectural description and dynamic reconfiguration of the system whenever services are discovered or disappear from the execution environment, and **(ii)** an extended version of the run-time composition and adaptation engine presented in [10], to work with the new model. Most of the aforementioned approaches in software adaptation build adaptors for the whole system, which is a costly process. Adaptor generation relies on the specific number of services involved in the system, and the adaptor has to be recomputed each time a new entity is taken into account. Therefore, these static approaches are not suitable in our problem context. Our run-time composition engine enables service adaptation without the need of generating adaptors, and has been extended in this work to ensure user-defined properties of the composition among services.

The rest of this paper is structured as follows. Section 2 presents our service model. Then, Section 3 describes how to achieve self-adaptation of service-based systems based on our model, including analysis of stable state reachability and a description of the run-time execution platform required. Next, Section 4 describes how temporal formulas can be used to enforce specific constraints in the composition of services. Finally, Section 5 compares our approach with the related work in the field, and Section 6 recalls the contributions of this paper and draws up its conclusions.

2 Model of Services

We propose a service interface model which includes both a signature, and a behavioural interface.

Signature. Set of required and provided operations signatures.

Definition 1 (Operation Signature) *An* Operation Signature *is the name of an operation, together with its arguments and return types.*

Behavioural Interface. Consists of: **(i)** a protocol description (STS); **(ii)** a set of generic correspondences between operations or *vectors*; and in some cases, **(iii)** a set of constraints, expressed as temporal formulas, to be satisfied by the composition of the service with the rest of the system.

Protocol description. Protocols are represented by means of *Symbolic Transition Systems* (STSs) [20]. This formal model has been chosen because it is simple, and it can be easily derived from existing implementation platforms' languages (see for instance [16, 33, 15] where such abstractions for Web services were used for verification, composition or adaptation purposes). Communication between services is represented using *events* relative to the emission (!) and reception (?) of *messages* which correspond to operation calls. Events may come with a set of data terms with types that respect the operation signatures. In our model, a *label* is either the internal action *tau* or a tuple (M, D, PL) where M is the message name, D stands for the direction (!,?), and PL is either a list of data terms if the message corresponds to an emission, or a list of variables if the message is a reception.

Definition 2 (STS) *An STS is a tuple* (A, S, I, B, T) *where: A is an alphabet that corresponds to message events relative to the service's provided and required operations,* S *is a set of states,* $I \in S$ *is the initial state,* $B \in S$ *are stable states, and* $T \in S \times A \times S$ *is the transition function. A stable state is one in which the service is not engaged in any transactions and can be removed from the current system configuration.*

Generic Correspondences. Adaptor generation approaches rely on a description of the adaptations to be performed, usually referred as *adaptation contracts*. These contracts express correspondences between operations names, or parameter types and ordering on the different interfaces. However, adaptation contracts can be produced only once the description of the different interfaces is known, since specific information such as operation names or parameter types is required. Unfortunately, in ubiquitous environments, this information is only available when the particular service is available in the system. Specifically, when we are describing the protocol of a service to be reused in such systems, we know what the required operations are, but we just do not know their specific name yet, or which service is going to provide them.

In order to overcome this situation, we assume the existence of a common *service ontology* that describes the properties and capabilities of services in unambiguous, computer-interpretable form. We also assume that services available in the system's environment are exposed using this service ontology, which will be used as a reference to relate service behaviour using elements of the specific domain. For simplicity, instead of using any of the emerging standards for the semantic description of services, we will use in the remainder of this paper a notation for this service ontology which abstracts away specific notations for service descriptions such as OWL-S[2] in Web services, or the Rosettanet[3] common standards in the case of e-business services:

Definition 3 (Abstract Operation Signature) *An* abstract operation signature *is the name of a generic operation, together with its arguments and return types defined within the context of a service ontology.*

Definition 4 (Abstract Role) *We define an* abstract role *as a set of abstract operation signatures associated with a common task or goal.*

In order to solve the aforestated problem, our model of interface contains generic correspondences of how labels in the protocol description (STS) are related with generic operations described by the service ontology (instead of directly relating labels between STSs like traditional adaptation contracts normally do). To do so, we rely on *synchronization vectors* [5] (or vector for short), which denote communication between different services, where each event appearing in one vector is executed by one service and the overall result corresponds to an interaction between all the involved services.

Definition 5 (Vector) *A vector for a service STS* (A, S, I, B, T) *and an abstract role, is a couple* $\langle e_l, e_r \rangle$ *where* e_l *is a label term for A, and* e_r *is an abstract label term for an abstract role. A label term t contains the name of the operation, a direction, and as many untyped fresh names as elements in the argument type list. To identify messages in a vector, label terms are prefixed by a service identifier. An* abstract label term *is defined in the context of an abstract role, instead of a service interface.*

Definition 6 (Vector Instance) *A vector instance for a pair of service STSs* $(A_l, S_l, I_l, B_l, T_l)$ *and* $(A_r, S_r, I_r, B_r, T_r)$ *is a couple* $\langle e_l, e_r \rangle$ *where* e_l, e_r *are label terms in* A_l *and* A_r*, respectively. To identify messages in a vector instance, label terms are prefixed by a service identifier, e.g.,* $\langle s_1 : comm!x, s_2 : comm?y \rangle$.

Vectors express correspondences between messages, like bindings between ports, or connectors, in architectural descriptions [19]. Furthermore, variables are used as placeholders in message parameters. The same variable name

[2] http://www.daml.org/services/owl-s/1.0/

[3] http://www.rosettanet.org/

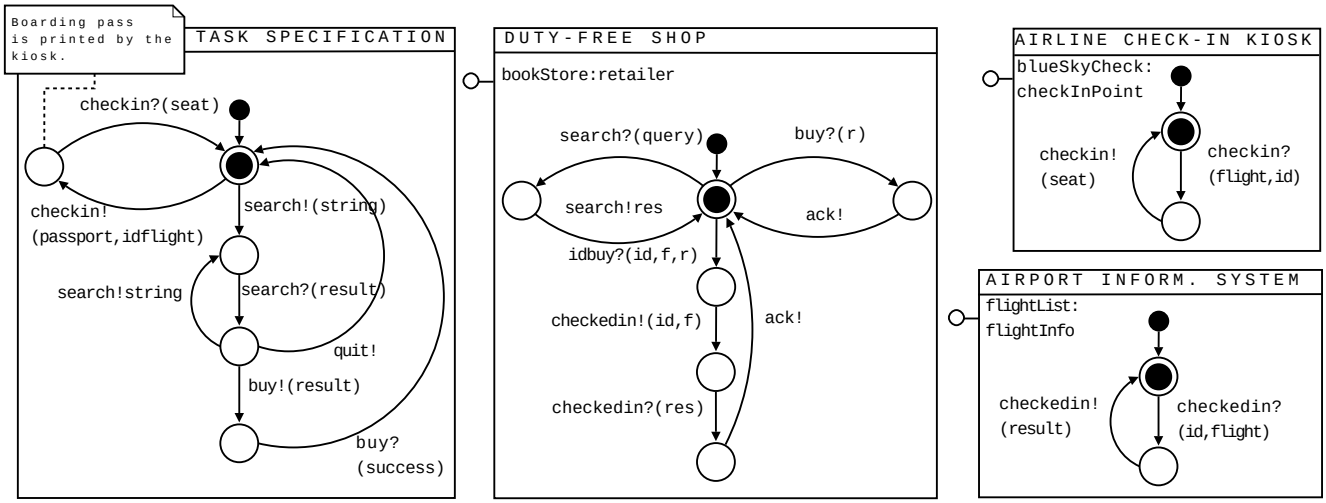

Figure 1. Service protocols for our running example. Initial and stable states are respectively noted in STSs using bullet arrows and darkened states

appearing in different labels (possibly in different vectors) enables the relation of sent and received arguments of messages. Vectors can be either written by hand, obtained from a graphical description of the architecture built by the designer, or automatically generated in some specific cases [28].

Constraints. Last but not least, interfaces may also add some constraints on the application order of its own vectors. This is specified in our approach using the LTL modal temporal logic [14]. This part of interface descriptions will be explained in further detail in Section 4.

Example. Figure 1 depicts the protocols for the different services and the user task specification involved in the example presented in the introduction. For space reasons, we describe service interfaces only with their STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels. If we take for instance the user task specification protocol, we can observe that it can initially either request a `checkin!`, receiving a response with the seat number assigned on the flight, or perform a `search!` on a shop's catalog. After that, the client can perform an arbitrary number of searchs, and then return to the initial state either buying an item (`buy!`), or aborting the transaction (`quit!`).

Table 2 lists the abstract roles fulfilled by each of the services in our example, along with the abstract operation signatures they contain. It is worth observing in Figure 1 that each service makes explicit the set of abstract roles they fulfill (only one on each service in our example). For instance, all check-in kiosks at the airport fulfill the `checkInPoint` abstract role. In particular, the kiosk in our example contains a role `blueSkyCheck` which fulfills the aforementioned abstract role.

Table 3 includes the correspondences between STS labels in each of the services, and abstract operations defined in abstract roles. For instance, $v_{checkin}$ in the airport information system, establishes that any requests for passenger information `passengerInFlight` in the abstract role `flightInfo`, will be received by `checkedin` in the service protocol. Likewise, the table contains a set of user-defined properties for the client task specification in the example which must be satisfied by the composition of the service with its environment. The purpose and specification of these properties will be detailed in Section 4.

3 Service Behavioural Self-Adaptation

In this section, we present our approach to build dynamically a system with services entering and leaving at runtime (see an overview in Figure 4). At this stage, we do not consider service interfaces equipped with temporal formulas, which will be presented in Section 4. Our proposal is completely automated, and guided by a user requirements description. Three main tasks have to be fulfilled every time a new entity appears or leaves: (i) instantiation of vectors, (ii) checking stable state reachability, (iii) run-time execution of the involved services.

Abstract Role	Operation	Argument Types	Return Val.
checkInPoint	checkIn	passportId, flightId	seatCode
flightInfo	passengerInFlight	passportId, flightId	boolean
retailer	search	searchString	itemId, price
	sale	itemId	boolean
	taxFreeSale	passportId, flightId, itemId	boolean
..			

Figure 2. Abstract role definitions in the airport example

USER TASK SPECIFICATION (U)
$v_{checkin}$ = ⟨checkin!(pid,fid); checkInPoint.checkIn?(pid,fid)⟩ $v_{checkinResponse}$ = ⟨checkin?(sid); checkInPoint.checkIn!(sid)⟩ v_{search} = ⟨search!(str); retailer.search?(str)⟩ $v_{searchResponse}$ = ⟨search?(iid); retailer.search!(iid,p)⟩ v_{sale} = ⟨buy!(iid); retailer.sale?(iid)⟩ $v_{taxFreeSale}$ = ⟨buy!(iid); retailer.taxFreeSale?(pid,fid,iid)⟩ $v_{saleResponse}$ = ⟨buy?(r); retailer.sale!(r)⟩ v_{quit} = ⟨quit!; process.stop?⟩ ...
$\Diamond v_{checkin}$ (Check-in will eventually happen) $\Box(v_{taxFreeSale} \rightarrow \neg\Diamond v_{checkin})$ (Tax Free sales will always happen after Check-in) $\Box(v_{checkin} \rightarrow \neg\Diamond v_{checkin})$ (Check-in will happen only once)
AIRLINE CHECK-IN KIOSK (K)
$v_{checkinRequest}$ = ⟨checkin?(fid,pid); checkInPoint.checkIn!(pid,fid)⟩ $v_{checkinResponse}$ = ⟨checkin!(sid); checkInPoint.checkIn?(sid)⟩
AIRPORT INFORMATION SYSTEM (F)
$v_{infoRequest}$ = ⟨checkedin?(pid,fid); flightInfo.passengerInFlight!(pid,fid)⟩ $v_{infoResponse}$ = ⟨checkedin!(sid); flightInfo.passengerInFlight?(sid)⟩
DUTY-FREE SHOP (S)
v_{search} = ⟨search?(s); retailer.search!(s)⟩ $v_{searchResponse}$ = ⟨search!(r); retailer.search?(r,p)⟩ $v_{purchase}$ = ⟨buy?(iid); retailer.purchase!(iid)⟩ $v_{taxFreePurchase}$ = ⟨idbuy?(pid,fid,iid); retailer.taxFreeSale?(pid,fid,iid)⟩ $v_{checkin}$ = ⟨hascheckedin!(pid,fid); flightInfo.passengerInFlight?(pid,fid)⟩ $v_{checkinResponse}$ = ⟨hascheckedin?(sid); flightInfo.passengerInFlight!(sid)⟩ v_{ack} = ⟨ack!; retailer.taxFreeSale?(r)⟩ $v_{taxFreeAck}$ = ⟨ack!; retailer.sale?(r)⟩

Figure 3. Service vector and property definitions for the airport example

3.1 Vector Instantiation

In this section, we describe how a set of vectors from several service interfaces are instantiated by using abstract operation signatures. For each vector v, we extract all the other vectors including abstract label terms corresponding to the same abstract operation signatures. A vector is instantiated as many times as there are possible combinations *wrt.* the set of available vectors.

More concretely, an instantiation of v is obtained in two steps: **(i)** finding a set of matching vectors V_m (sharing abstract label terms with v, although with opposite direction in communication), and **(ii)** aggregating in a new set of vector instances V_{inst} the non-abstract label term of v with all the non-abstract label terms appearing in each of the vectors in V_m. The algorithm keeps track on already instantiated vectors to avoid repeating instantiations.

Example. We consider a scenario in our running example where the user enters the airport and walks into the check-in area. Once there, the airline check-in kiosk service becomes available, and vectors both on the user task specification and the check-in service interface must be used in order to instantiate the vectors which will make the interaction of both partners possible. As it can be observed in Table 3, the only abstract label terms shared by both partners correspond to `checkInPoint.checkIn`:

USER TASK SPECIFICATION

$v_{checkin}$
⟨checkin!(pid, fid); checkInPoint.checkIn?(pid, fid)⟩

$v_{checkinResponse}$
⟨checkin?(sid); checkInPoint.checkIn!(sid)⟩

...

AIRLINE CHECK-IN KIOSK

$v_{checkinRequest}$
⟨checkin?(fid, pid); checkInPoint.checkIn!(pid, fid)⟩

$v_{checkinResponse}$
⟨checkin!(sid); passenger.checkIn?(sid)⟩

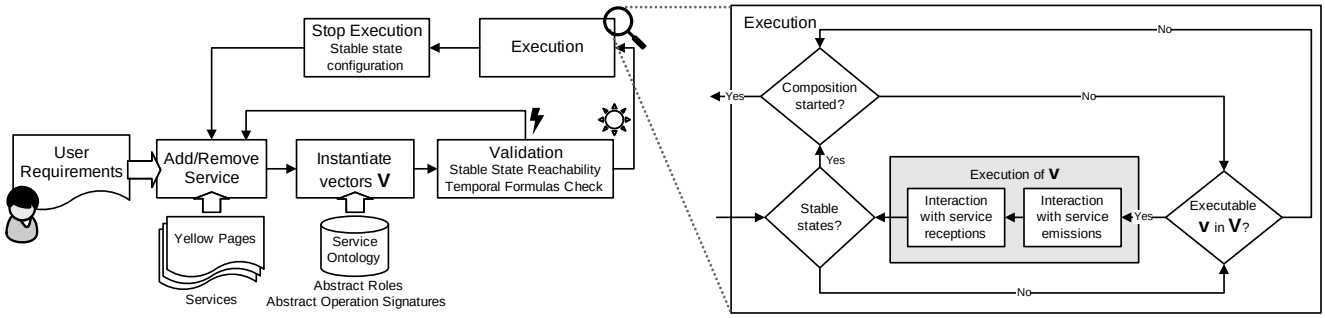

Figure 4. Overview of our approach

VECTOR INSTANCES
v_{insta}
$\langle \mathtt{u} : \mathtt{checkin}!(\mathtt{p}, \mathtt{f}); \mathtt{k} : \mathtt{checkin}?(\mathtt{f}, \mathtt{p})\rangle$
v_{instb}
$\langle \mathtt{u} : \mathtt{checkin}?(\mathtt{s}); \mathtt{k} : \mathtt{checkin}!(\mathtt{s})\rangle$

As it can be observed above, each vector instance includes a set of STS label terms prefixed by an identifier of the service. In this case, $v_{checkin}$ on the task specification has been matched with $v_{checkinRequest}$ on the kiosk, resulting on v_{insta}. Vector instance names are not relevant, since they are only used for run-time execution of the system. It is worth noticing that fresh names p,f, and s are used as placeholders for correct parameter exchange. These names are placed in the vector instance taking as reference the order of parameters in the shared abstract label term. Notice the inverted order of these names in the different label terms of v_{insta}.

3.2 Stable State Reachability

Once we have described interfaces and the process of vector instantiation, in this section we sketch two solutions to compute the reachability analysis of stable states being given a set of service protocols, and a set of instantiated vectors. A stable state of the system is one, where each of the services in the system is on a stable state. It is only at this point that services can be incorporated or removed, and the system properly reconfigured.

A first solution is an *ad-hoc* search algorithm. In [10], we presented a depth-first search algorithm which seeks correct termination states, 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 either a final state is reached (end of the algorithm), or a deadlock state is found. In the latter case, the algorithm backtracks and tries a different path. We keep track of the already traversed states and use Floyd's cycle-finding algorithm [24][4] in order to avoid reach already visited states and remain indefinitely in cyclic paths. This solution is appropriate for systems of a moderate size (see [10] for details).

However, in the context of pervasive systems where a large number of services have to interact or feature protocols with a large number of states and transitions, a second solution is to use an informed search algorithm in order to find potential solutions efficiently. Specifically, the A* algorithm, is a particular best-first search strategy which determines the mimimum cost path from a given state of the system to a goal state by expanding the most promising candidate paths first. However, guidance information for this search is required. This is achieved by defining a heuristic estimation function of the cost of arriving from the current state of the services to a global stable state in the composition. In our particular case, we use a heuristic estimation based on the minimum distance from the current state of the system to a global stable state (see [8] for details).

3.3 Run-time Execution

Once vectors instantiated and validation achieved to be sure that the system can be run and will reach a correct termination state (global stability), execution of the system can be launched. In this section, we present a run-time engine that executes a system involving a set of services using vectors as their interaction constraints. We promote a dynamic execution of vectors instead of the execution of an adaptor generated statically from vectors, because adaptor generation is costly and algorithms are exponential [12, 32].

The application of the composition specification (vectors) can lead the system constituted of the involved services into deadlocking situations. This can be caused by a missing service, a missing interaction, or a possible execution scenario that makes the system fail. Indeed, the composition specification is an abstract description of how services 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

[4] Floyd never published his cycle-finding algorithm. It was first presented by Knuth in the referenced book.

reach a stable 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 composition and adaptation approaches [21, 29]. Therefore, before applying a vector, we check that after the application of this vector, there exists at least one reachable stable state for the whole system (*i.e.*, all services are in a stable state). Thus, our dynamic composition engine will prevent the system to end up in deadlocking situations. This check is achieved using techniques presented in Section 3.2.

Figure 4 (right) sketches the operation of the dynamic composition engine. More details about that and ideas of how these techniques can be implemented using Dynamic-AOP (Aspect-Oriented Programming) can be found in [10].

Example. The passenger walks into the airport and reaches the check-in kiosk. At this point vectors for client and the kiosk check-in service are instanced. In Figure 5 we can observe how the check-in process is executed (v_{insta},v_{instb}) and the system reaches a stable state. At this point, services can be added to or removed from the system. The user now enters a duty free shop. The check-in service is no longer accessible, but the shop's service (connected to the airport information system) becomes available. The vector instantiation process is performed again. It is worth noticing that this time no vectors for non-tax free sales are instanced, since the user task specification on the client does not contain any vector definitions for that kind of transaction. The user searches the catalog (one or more times) for products (v_{instc},v_{instd}), and then performs a purchase(v_{inste}). The shop contacts the airport information system to confirm that the passenger is checked-in on a flight (v_{instf},v_{instg}), and then acknowledges the end of the transaction (v_{insth}).

4 Self-Adaptation with Temporal Formulas

In this section we describe how the interaction of different services can be constrained at run-time by defining interesting properties of the composition which must be satisfied by every possible execution trace of the system. To express such properties, we make use of linear temporal logic (LTL). In particular, we use the next-free variant of LTL (LTL-X)[5], guaranteed to be insensitive to stuttering [13].

In our approach, LTL atomic propositions correspond to vector labels. Since our run-time execution engine executes vectors one after the other in a sequential fashion to make the system evolve, we can assume that the execution of v is synonymous to the proposition v in a temporal formula. Hence, given a finite set of atomic propositions υ, formulas are constructed inductively as:

[5]LTL-X denotes the class of LTL formulas without the *n*ext temporal operator. In the rest of this paper, the use of LTL-X is implied whenever LTL is mentioned.

Propositions Every $\phi \in \upsilon$ is a formula.

Boolean operators Given the formulas ϕ and ψ: $\phi \rightarrow \psi$, $\phi \wedge \psi$, $\phi \vee \psi$, and $\neg\phi$ are also formulas.

Temporal operators given the formulas ϕ and ψ: $\phi U \psi$ is also a formula (strong until). The following abbreviations are used: Eventually ($\Diamond\phi = TRUE\ U\phi$) and Always ($\Box\phi = \neg\Diamond\neg\phi$).

We use LTL finite-trace semantics similar to the one defined in [17], commonly used in runtime verification. An interpretation of an LTL formula is a finite word $w = x_0 x_1 \ldots x_n$ over 2^{υ}, where at some time point $i \in N$ a proposition ϕ is true iff (if and only if) $\phi \in x_i$. We express as w_i the suffix of w starting at i. The finite-trace semantics of LTL is defined as:

Propositions For $\phi \in \upsilon, w \models \phi$ iff $\phi \in x_0$.

Boolean operators Given the formulas:

- $w \models \neg\phi$ iff not $w \models \phi$
- $w \models \phi \wedge \psi$ iff $w \models \phi$ and $w \models \psi$
- $w \models \phi \vee \psi$ iff $w \models \phi$ or $w \models \psi$

Temporal operators $w \models \phi U \psi$ iff there exists $0 \leq i \leq n$ such that $w_i \models \psi$ and for all $0 \leq j < i, w_j \models \phi$.

4.1 Defining Composition Properties Using Temporal Logic

When the designer is defining how a service must interact with its environment, it is interesting to specify:

Safety properties declaring what should not happen while the service is interacting with the rest of the system. Hence, no state of the execution path of the system should violate the property. Following with our running example, the user can specify on its client a safety property stating that the check-in process should be performed only once at the most, by writing the following formula:

$\Box(v_{checkin} \rightarrow \neg\Diamond v_{checkin})$ (Check-in will happen no more than once)

Liveness properties stating what should eventually happen while the service interacts with the rest of the system. As a consequence, the property must hold at some point of the execution path to be satisfied. In our example, an interesting liveness property for the user is ensuring that the check-in process is going to be performed at some point of the execution:

$\Diamond v_{checkin}$ (Check-in will eventually happen)

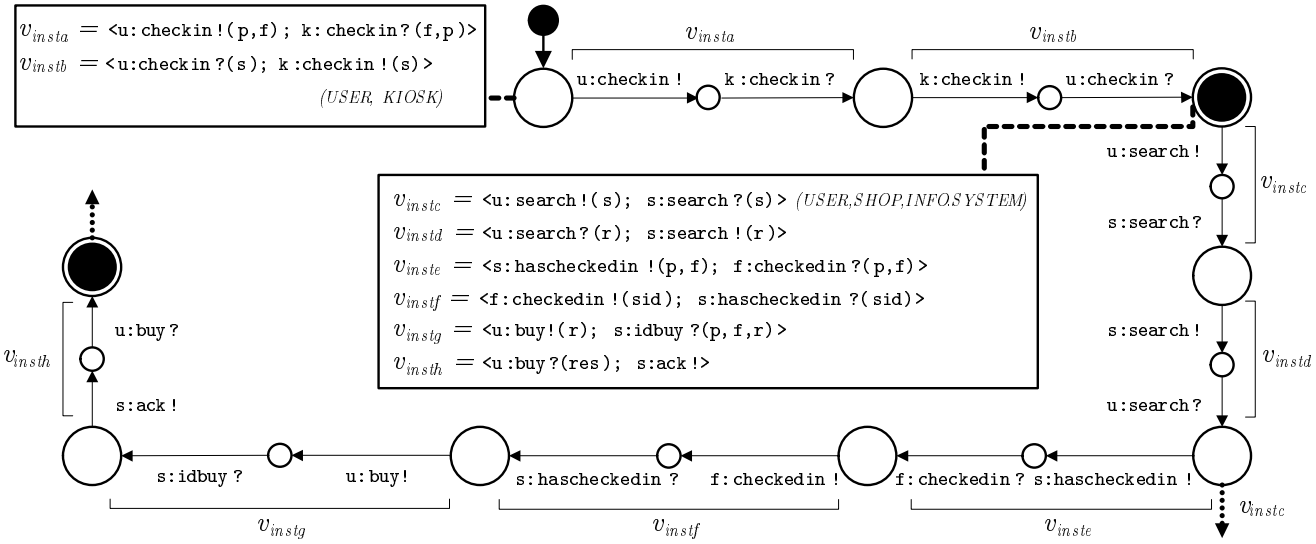

Figure 5. Execution trace of the airport example

4.2 Evaluating Properties at Run-time

Once we have defined the properties which should be respected by the composition on each of the interacting services, these must be evaluated on the set of potential execution paths of the system. Each one of those paths is formed by two parts: **(i)** Transaction history path. Is the sequence of all vectors executed before the current state in which the particular service has been involved. **(ii)** One of the candidate branches explored in the search for stable states described in Section 3.2. This search starts from the current state of the execution and only vectors involving the service which contains the property are taken into account.

Stable state search is performed on potential execution traces formed by vector instances, whereas the execution paths used for property evaluation are sequences of vectors local to the service. Although both stable state reachability and property satisfiability depend on the global behaviour of the system, the latter must be performed from a point of view local to the service, since properties are defined on its protocol and are independent from the rest of the system. To obtain both the transaction history path and potential local execution paths for each of the services, we trace back the correspondence between vector instances and local vector definitions. Hence, vector instance v_{insta} from our example in Section 3.1 would correspond to $v_{checkin}$ from the point of view of the user specification task, and $v_{checkinRequest}$ in the case of the check-in kiosk.

To evaluate the properties on the set of execution paths, each LTL formula is translated into an automaton (see [17] for details) which accepts a finite sequence of symbols (vector labels local to the service). Hence, if the automaton reaches an acceptance state after having received a sequence of vector labels as input (*i.e.*, an execution path), the property is satisfied. In particular, we use the evaluation of properties in different ways depending on what they specify:

Safety Properties For any safety property of the form $\Box\phi$ ($\neg\Diamond\neg\phi$), we build the automaton for $\Diamond\neg\phi$. In such a way, it is possible to prune the search conveniently whenever we arrive to an acceptance state in the automaton (the safety property has been violated).

Liveness Properties For any safety property of the form $\Diamond\phi$, we build its automaton, and only check if the property is satisfied on the corresponding trace once we arrive to a stable state found by our search algorithm presented in Section 3. If the property is not satisfied, the algorithm keeps searching for stable states where the property is satisfied.

Integrating property evaluation into our search algorithm in the aforedescribed fashion saves further exploration for stable states which, although reachable, would not satisfy the properties of the system.

Example. Considering the system execution scenario described in Section 3.3, we focus on the property included in the client stating that check-in must happen only once ($\Box(v_{checkin} \rightarrow \neg\Diamond v_{checkin})$). In this case, vector $v_{checkin}$ corresponds to vector instance v_{insta}. As it can be observed in Figure 6, the transaction history path consists on the sequence v_{insta},v_{instb}. Before executing another vector instance, our search algorithm starts exploring alternative branches trying to find stable states. As it can be observed, the application of the same sequence of vector instances (v_{insta},v_{instb}) would lead to a stable state. However, the second application of v_{insta} leads the automaton ($\Diamond\neg(v_{checkin} \rightarrow \neg\Diamond v_{checkin})$) for

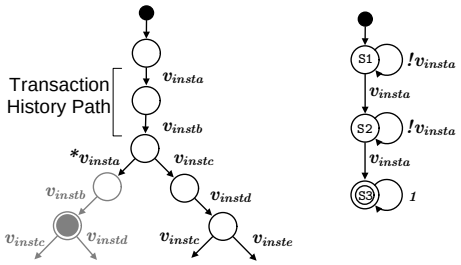

Figure 6. Safety property violation example

the property to the acceptance state S3. At this point, the property is violated and hence the search continues through the alternative branch starting with v_{instC}.

5 Related Work

In this section, we will compare successively our approach with some proposals in related areas, namely software adaptation, dynamic reconfiguration of software, and self-adaptive systems.

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 [34, 21, 6, 7, 12] 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. However, all these approaches assume that all the components involved in the system are known at design-time, and no new entity can be added or removed dynamically. A recent work aims at building adaptors incrementally [32]. 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. Here, this incremental construction of the system-to-be takes place at design-time or off-line.

Dynamic reconfiguration [31] is not a new topic and many solutions have already been proposed dealing with distributed systems and software architectures [25, 26], graph transformation [1, 36] or metamodelling [23, 30]. For instance, Kramer and Magee [26] studied how changes are applied dynamically to system composed of components and their interconnections. To preserve the integrity of the system, they propose a notion of *quiescent* portions during which changes can be performed. Analysis techniques (animation and reachability) are used in this approach to check that changes do not violate properties to be ensured by the system. In our proposal, we do not allow the modification of the components at hand, but permit to add and remove them at run-time. In addition, we assume the architecture not defined by the designer but inferred automatically from the service interfaces. However, the notion of stable states we defined in this paper is very similar to the quiescent portions introduced in [26].

Self-adaptation is an emerging and very promising topic for systems running in dynamically changing environments. Several recent proposals tackle different issues in this area, such as software architecture self-adaptation [35], service replacement at run-time [22], or adaptive systems modelling issues [18]. In [35] for instance, the authors propose using planning techniques to build new configurations of a system. Reactive plans are generated with a planning tool from high-level goals given by the user. Each plan defines a set of conditional rules that indicate what components are required to execute the plan. Our solution based on vectors and temporal formulas can be considered as an alternative to use planning techniques. The main difference is that this approach requires pre-defined global goal given by a designer whereas in our proposal such goals are dynamically provided by the user.

6 Concluding Remarks

In this paper, we have presented our proposal to self-adaptation of services specified with rich interfaces (protocols, vectors, and possibly some temporal formulas). The high expressiveness of service interfaces allow to completely automate the reconfiguration of the system at run-time (removal or arrival of new services). We have explained how the different steps of our approach are applied, and how correctness of the execution is ensured by using stable states and run-time evaluation of service properties. We have illustrated these ideas on a real-world example.

As far as future works are concerned, our main goal is to include in our ITACA toolbox[6] [9] our model of service interfaces extended with vectors and temporal formulas, and implement an extension of our dynamic execution engine to ensure user defined properties on the composition.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN), and project P06-TIC-02250 funded by the *Junta de Andalucía*.

[6]ITACA is a toolbox under implementation at the University of Málaga dedicated to the automatic composition and adaptation of services accessed through their behavioural interfaces. Accessible at http://itaca.gisum.uma.es

References

[1] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, volume 2621 of *LNCS*, pages 37–51. Springer, 2003.

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

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

[4] 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.

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

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

[7] A. Brogi, C. Canal, and E. Pimentel. Component Adaptation Through Flexible Subservicing. *Sci. Comput. Programming*, 63(1):39–56, 2006.

[8] J. Cámara, C. Canal, and G. Salaün. Multiple concern adaptation for run-time composition in context-aware systems. *Electr. Notes Theor. Comput. Sci*, 215:111–130, 2008.

[9] J. Cámara, J. A. Martí, G. Salaün, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE'09*, 2009. To appear.

[10] J. Cámara, G. Salaün, and C. Canal. Composition and run-time adaptation of mismatching behavioural interfaces. *Journal of Universal Computer Science*, 14(13):2182–2211, 2008.

[11] C. Canal, J. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1), 2006. Special Issue on WCAT'04.

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

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[14] E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

[15] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.

[16] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.

[17] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. IEEE Computer Society, 2001.

[18] H. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Proc. of ECBS'08*, pages 36–45. IEEE Computer Society, 2008.

[19] S. Haddad and P. Poizat. Transactional Reduction of Component Compositions. In *Proc. of FORTE'07*, volume 4574 of *LNCS*, pages 341–357. Springer, 2007.

[20] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

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

[22] F. Irmert, T. Fischer, and K. Meyer-Wegener. Runtime Adaptation in a Service-Oriented Component Model. In *Proc. of SEAMS'08 (held with ICSE'08)*. ACM Press, 2008.

[23] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, volume 3107 of *LNCS*, pages 264–273. Springer, 2004.

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

[25] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[26] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

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

[28] J. Martín and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, ENTCS, 2008. To appear.

[29] R. Mateescu, P. Poizat, and G. Salaün. Behavioral Adaptation of Component Compositions based on Process Algebra Encodings. In *Proc. of ASE'07*. IEEE Computer Society, 2007.

[30] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.

[31] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM Press, 1996.

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

[33] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

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

[35] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Plan-Directed Architectural Change for Autonomous Systems. In *Proc. of SAVCBS'07 (held with FSE'07)*. ACM Press, 2007.

[36] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE 2001*, pages 21–32. ACM Press, 2001.

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