

Towards Self-Adaptation in Dynamic Service Ecosystems

Javier Cámara, Carlos Canal, y Gwen Salaün

Department of Computer Science, University of Málaga, Spain
{jcamara,canal,salaun}@lcc.uma.es

Resumen. 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 components and goals of the system. This approach is suitable for *closed systems*, where constituent components are well known at design time. However, there are many situations in which software systems are *open*, and new components or services may dynamically join or leave the context of the system at any given moment. This results in a lack of a predefined description of the system's architecture and goals. In such open systems, new adaptability problems arise, such as the connection and disconnection of unforeseen components to an already running system; interoperability issues among third-party components not specifically designed to interact with each other; and 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 open systems, dynamically reconfiguring the system whenever services are added to or removed from it.

Palabras clave: Service Composition, Run-time Adaptation, Open Systems, Behavioural Interface

1 Introduction

The proliferation of software services and ubiquitous computing, as well as other factors, such as the increasing need of communicating systems spanning across both organizational and physical boundaries, are shaping a panorama where the flow of information and connectivity between previously unacquainted systems and devices is desirable –if not necessary. However, in most cases software services are not designed to interoperate with each other, since the different scenarios in which a particular service may be used cannot be envisioned *a priori* by their designers. The need to discover and access services as users move from one location to another and the conditions of the environment change quickly over time, is one crucial requirement in the design of the system, its constituent components, and also the services they are accessing and using during their operation. Hence, all these elements must be able to achieve a certain degree of adaptation in order to successfully interoperate with each other.

Software Adaptation [10, 34] provides the abstractions and mechanisms to enable services with mismatching interfaces to interoperate at different levels (*i.e.*, signature, protocol, service and semantics). Continuous change and short-term association among software entities demand the automation of this adaptation process.

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 components and goals of the system (usually following a top-down

approach), or on the definition of local rules for the different elements of the system, which results in an emergent self-organizing behaviour (hence following a bottom-up approach). Both these approaches are suitable for *closed systems*, that is, those whose constituent components are well known at design time, or where there is no need to explicitly represent the goals of the system. Industrial control systems, robotics, or autonomous aircrafts are typical domains in which software systems are inherently closed, due to their close dependency to hardware parts.

However, there are many situations in which software systems are *open*, and the changes in their execution environment are directly subject to the availability of a particular service, which may join or leave the context of the system at any given moment. These systems lack a predefined description of its architecture and components, and even of its goals. Paradigmatic examples of such open systems can be found in ubiquitous computing, or in dynamic web service discovery and composition. In such open systems, new adaptability problems arise, such as the connection and disconnection of a new software component to an already running system; or how to solve interoperability issues among third-party components not specifically designed to interact with each other, just to name a few.

In this paper we present our approach for the composition and resolution of potential interoperability issues among services —as those presented above—, dynamically reconfiguring the system whenever services are added to or removed from it. Particularly, we focus on the protocol or behavioral level, currently acknowledged as one of the most relevant research directions in software adaptation [2–4, 6, 23, 34]. Mismatch at this level involves undesirable system behavior such as *deadlock*, or incorrect termination states, mainly derived from the order of the messages exchanged.

The rest of this paper is structured as follows. Section 2 presents our service model, including the definitions of behavioral signatures, (consisting of roles and operations, and described by means of symbolic transition systems) and vectors for stating correspondences between service signatures. 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. Finally, Section 4 compares our approach with the related work in the field, and Section 5 recalls the contributions of this paper and draws up its conclusions.

2 Model of Services

First of all, we assume the existence of a common language with constructs for describing the properties and capabilities of services in unambiguous, computer-interpretable form. Specifically, we focus on the description of the different operations available in the different service interfaces. We assume that services available in the system's environment are exposed using this common language, which will be used as a reference to relate service behaviour using elements of the specific domain. Instead of using any of the emerging standards for the semantic description of services, we will use a notation for this common language which abstracts away specific notations for service descriptions such as OWL-S¹ in Web services, or the Rosettanet² common standards in the case of e-business.

Definition 1 (Role). *We define a role as a set of operations associated with a particular task or goal in the context of a service interface. In the case of a set of abstract operations associated with a common task in the context of our common language, we will refer to it as an abstract role.*

¹ <http://www.daml.org/services/owl-s/1.0/>

² <http://www.rosettanet.org/>

Definition 2 (Operation Signature). *An Operation Signature is the name of an operation, together with its arguments and return types. If the operation signature is defined within the context of an abstract role, rather than on a service interface, we will refer to it as an abstract operation signature.*

Service interfaces are equipped with both a set of required and provided operations signatures, and a behavioural interface. The behavioural interface includes: **(i)** a protocol description; **(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.

Protocols are represented by means of *Symbolic Transition Systems* (STSs) [16]. This formal model has been chosen because it is simple, and it can be easily derived from existing implementation platforms' languages, see for instance [13, 30, 12] 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 (denoted using $!$) and reception (denoted using $?$) of *messages* which correspond to operation calls. Events may come with a set of data terms whose types 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 3 (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 execution of the service terminates correctly.*

Furthermore, interfaces contain generic correspondences on how their labels (messages) are related with abstract operation signatures described on abstract roles. To do so, we rely on *synchronization vectors* [5] (or vector for short), which denotes 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. A vector does not require interactions to occur on the same names of events³.

Definition 4 (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 operation signature. 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 corresponds to an abstract role instead of a service interface, and it is prefixed by its name.*

Definition 5 (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 [15]. Furthermore, variables are used as placeholders in message parameters. The same variable name appearing in different labels (possibly in different vectors) enables the relation of sent and received arguments of messages. Vectors

³ It is worth noticing that since interactions are expressed using vectors, the interaction semantics of STS is not used within our model.

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 [24].

Example. In order to illustrate our approach, we describe a running example which consists of a set of services described within the context of an airport. A traveler walks into the airport carrying a handheld device with a client containing a task specification based on the different services which may be required while staying at the airport. In Figure 1, we can observe that the user first needs to **checkin!** with the airline 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 user may enter any of the duty-free shops located at the airport, and locally **search!** for different items which can be purchased (**buy!**). The different shops are able to access the airport information system, which features a listing service able to check whether a passenger is checked-in on a particular flight, applying a tax exemption on the sale if this is the case. 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.

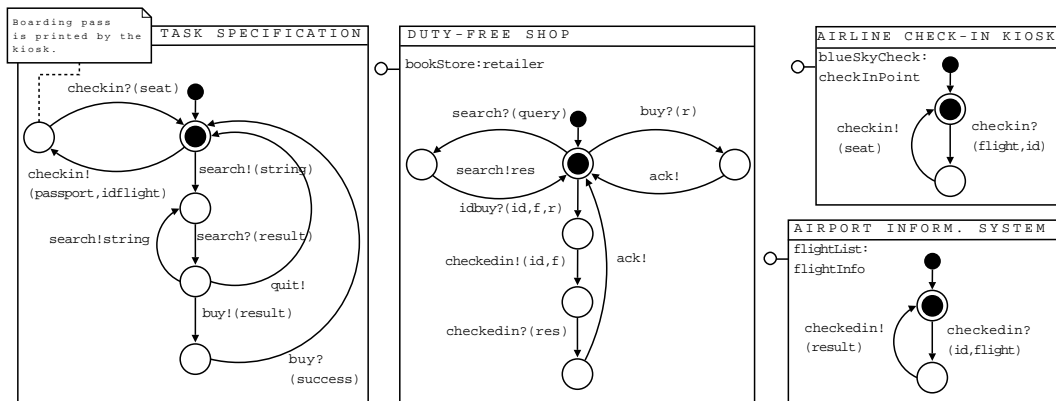


Fig. 1. Service protocols for our running example. Initial and stable states are respectively noted in STSs using bullet arrows and darkened states.

Table 1 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 2 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.

3 Self-Adaptation with Vectors

In this section, we present our approach to build dynamically a system with services entering and leaving at run-time (see an overview in Figure 2). Our proposal is completely automated,

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
..			

Tabla 1. Abstract role definitions in the airport example.

<p>USER TASK SPECIFICATION (U)</p> <p>$v_{checkin} = \langle \text{checkin!}(\text{pid}, \text{fid}); \text{checkInPoint.checkIn?}(\text{pid}, \text{fid}) \rangle$</p> <p>$v_{checkinResponse} = \langle \text{checkin?}(\text{sid}); \text{checkInPoint.checkIn!}(\text{sid}) \rangle$</p> <p>$v_{search} = \langle \text{search!}(\text{str}); \text{retailer.search?}(\text{str}) \rangle$</p> <p>$v_{searchResponse} = \langle \text{search?}(\text{iid}); \text{retailer.search!}(\text{iid}, \text{p}) \rangle$</p> <p>$v_{sale} = \langle \text{buy!}(\text{iid}); \text{retailer.sale?}(\text{iid}) \rangle$</p> <p>$v_{taxFreeSale} = \langle \text{buy!}(\text{iid}); \text{retailer.taxFreeSale?}(\text{pid}, \text{fid}, \text{iid}) \rangle$</p> <p>$v_{saleResponse} = \langle \text{buy?}(\text{r}); \text{retailer.sale!}(\text{r}) \rangle$</p> <p>...</p>
<p>AIRLINE CHECK-IN KIOSK (K)</p> <p>$v_{checkinRequest} = \langle \text{checkin?}(\text{fid}, \text{pid}); \text{checkInPoint.checkIn!}(\text{pid}, \text{fid}) \rangle$</p> <p>$v_{checkinResponse} = \langle \text{checkin!}(\text{sid}); \text{checkInPoint.checkIn?}(\text{sid}) \rangle$</p>
<p>AIRPORT INFORMATION SYSTEM (F)</p> <p>$v_{infoRequest} = \langle \text{checkedin?}(\text{pid}, \text{fid}); \text{flightInfo.passengerInFlight!}(\text{pid}, \text{fid}) \rangle$</p> <p>$v_{infoResponse} = \langle \text{checkedin!}(\text{sid}); \text{flightInfo.passengerInFlight?}(\text{sid}) \rangle$</p>
<p>DUTY-FREE SHOP (S)</p> <p>$v_{search} = \langle \text{search?}(\text{s}); \text{retailer.search!}(\text{s}) \rangle$</p> <p>$v_{searchResponse} = \langle \text{search!}(\text{r}); \text{retailer.search?}(\text{r}, \text{p}) \rangle$</p> <p>$v_{purchase} = \langle \text{buy?}(\text{iid}); \text{retailer.purchase!}(\text{iid}) \rangle$</p> <p>$v_{taxFreePurchase} = \langle \text{idbuy?}(\text{pid}, \text{fid}, \text{iid}); \text{retailer.taxFreeSale?}(\text{pid}, \text{fid}, \text{iid}) \rangle$</p> <p>$v_{checkin} = \langle \text{hascheckedin!}(\text{pid}, \text{fid}); \text{flightInfo.passengerInFlight?}(\text{pid}, \text{fid}) \rangle$</p> <p>$v_{checkinResponse} = \langle \text{hascheckedin?}(\text{sid}); \text{flightInfo.passengerInFlight!}(\text{sid}) \rangle$</p> <p>$v_{ack} = \langle \text{ack!}; \text{retailer.taxFreeSale?}(\text{r}) \rangle$</p> <p>$v_{taxFreeAck} = \langle \text{ack!}; \text{retailer.sale?}(\text{r}) \rangle$</p>

Tabla 2. Service vector definitions for the airport example.

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 states reachability, (iii) run-time execution of involved services. Note that reachability of global stable states conditions (point (ii) above) the execution launching, and is also used at run-time to execute only interactions for which such (global stable) states exist.

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

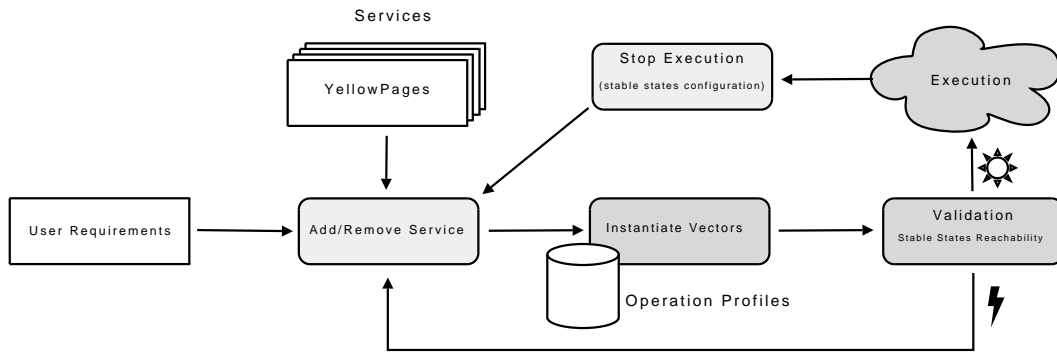


Fig. 2. Overview of our approach

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 2, the only abstract label terms shared by both partners correspond to `checkInPoint.checkIn`:

```

USER TASK SPECIFICATION
 $v_{checkin} = \langle \text{checkin!}(\text{pid}, \text{fid}); \text{checkInPoint.checkIn?}(\text{pid}, \text{fid}) \rangle$ 
 $v_{checkinResponse} = \langle \text{checkin?}(\text{sid}); \text{checkInPoint.checkIn!}(\text{sid}) \rangle$ 
...
AIRLINE CHECK-IN KIOSK
 $v_{checkinRequest} = \langle \text{checkin?}(\text{fid}, \text{pid}); \text{checkInPoint.checkIn!}(\text{pid}, \text{fid}) \rangle$ 
 $v_{checkinResponse} = \langle \text{checkin!}(\text{sid}); \text{passenger.checkIn?}(\text{sid}) \rangle$ 
VECTOR INSTANCES
 $v_{insta} = \langle \text{u:checkin!}(\text{p}, \text{f}); \text{k:checkin?}(\text{f}, \text{p}) \rangle$ 
 $v_{instb} = \langle \text{u:checkin?}(\text{s}); \text{k:checkin!}(\text{s}) \rangle$ 

```

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 parameter values. 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 [8], 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 [20]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 [8] 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. In [9] we presented a best-first search strategy which determines the minimum cost path from a given state of the system to a global stable state by expanding the most promising candidate paths first.

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 [11, 28].

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 reach a final state.

Since the composition specification is applied at run-time, it is not possible to apply the removal of deadlocks as a pre-processing as it is the case in static composition and adaptation approaches [17, 25]. Therefore, before applying a vector which belongs to the composition specification, we check that after the application of this vector, there exists at least one reachable stable state for the whole system (*i.e.*, all services and the composition specification are in a final state). Thus, our dynamic composition engine will prevent the system to end up in deadlocking situations, and will run only interactions that will make the system terminate. This check is achieved using techniques presented in Section 3.2.

We sketch in Figure 3 the main steps on which rely the dynamic composition engine. More details about that and ideas of how these techniques can be implemented in practice using Dynamic-AOP (Aspect-Oriented Programming) can be found in [8].

Example. The passenger walks into the airport and reaches the check-in kiosk. At this point vectors for the handheld device client and the kiosk check-in service are instanced. In Figure 4 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 walks towards the shopping area and 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

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

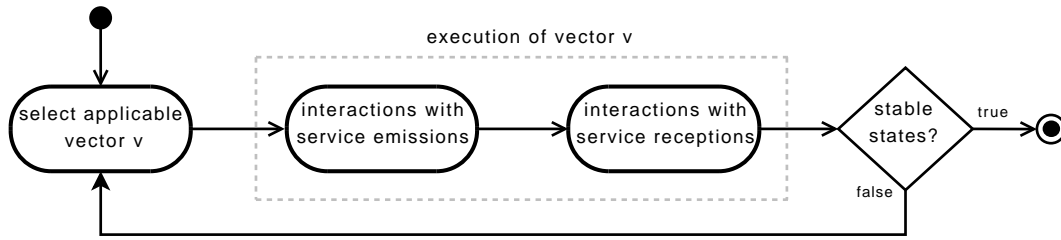


Fig. 3. Dynamic Composition Engine Overview

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}).

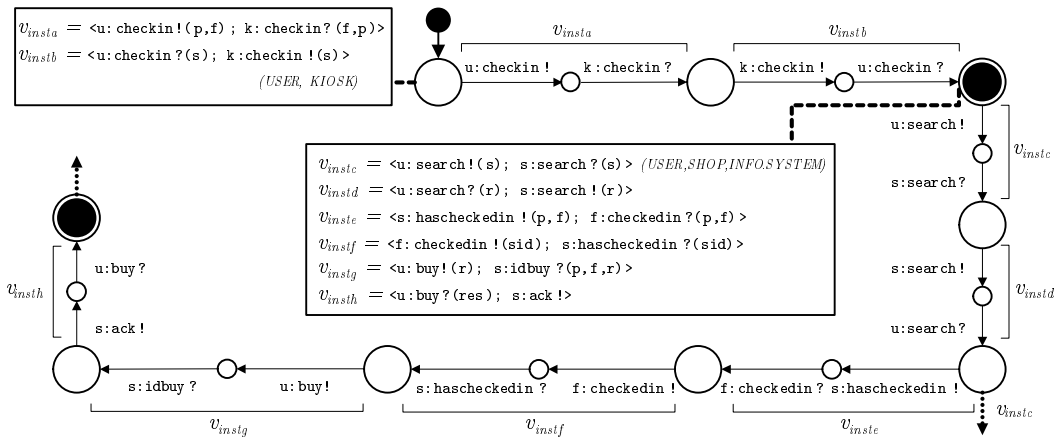


Fig. 4. Execution trace of the airport example.

4 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 [31, 17, 6, 7, 11] 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 [29]. 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 [27] is not a new topic and many solutions have already been proposed dealing with distributed systems and software architectures [21, 22], graph transformation [1, 33] or metamodelling [19, 26]. For instance, Kramer and Magee [22] 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 [22].

Self-adaptive systems and self-adaptation are emerging and very promising topics for systems running in dynamically changing environments. Several recent proposals tackle different issues in this area, such as software architecture self-adaptation [32], service replacement at run-time [18], or adaptive systems modelling issues [14]. In [32] 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 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 determined by service interfaces.

5 Concluding Remarks

In this paper, we have presented our proposal to self-adaptation of services specified with rich interfaces (protocols and vectors). 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 in this paper how the various steps of our approach apply, and how correctness of the execution is ensured by using stable state reachability evaluation at run-time. We have illustrated these ideas on a real-world example.

As far as future works are concerned, our main goal is to extend our ITACA toolbox⁵ to extend our model of service interfaces with abstract vectors and temporal formulas stating specific service goals for their composition, implement techniques to instantiate vectors from a set of interfaces, and implement an execution engine that would apply instantiated vectors while ensuring stable state reachability and formula validity.

Acknowledgements. This work has been partially supported by the CARESS project (TIN2007-67134), funded by the Spanish Ministry of Innovation and Science, and project P06-TIC-02250 funded by the Andalusian local Government.

⁵ ITACA is a toolbox we are implementing at the University of Málaga dedicated to the automatic composition and adaptation of services accessed through their behavioural interfaces.

Referencias

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.A. 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. *Science of Computer Programming*, 63(1):39–56, 2006.
8. J. Cámara, G. Salaün, and C. Canal. Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *Journal of Universal Computer Science*, 2008. To appear.
9. Javier Cámara, Gwen Salaün, and Carlos Canal. Multiple concern adaptation for run-time composition in context-aware systems. In *Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*, volume 215 of *Electronic Notes in Theoretical Computer Science*, pages 111–130. Elsevier, 2008.
10. C. Canal, J.M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1), 2006. Special Issue on WCAT'04.
11. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, *LNCS* 4037. Springer.
12. H. Foster, S. Uchitel, and J. Kramer. LTS-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.
13. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.
14. 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.
15. S. Haddad and P. Poizat. Transactional Reduction of Component Compositions. In *Proc. of FORTE'07*, volume 4574 of *LNCS*, pages 341–357. Springer, 2007.
16. A. Ingólfssdóttir and H. Lin. *A Symbolic Approach to Value-passing Processes*, pages 427–478. Handbook of Process Algebra. Elsevier, 2001.
17. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
18. 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.
19. 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.
20. D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.
21. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
22. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
23. J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour Analysis of Software Architectures*, pages 35–49. Kluwer Academic Publishers, 1999.
24. J.A. Martín and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, *ENTCS*, 2008. To appear.

25. 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.
26. J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.
27. N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM Press, 1996.
28. P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *LNCS*, pages 141–156. Springer, 2007.
29. P. Poizat and G. Salaün. Adaptation of Open Component-based Systems. In *Proc. of FMOODS'07*, volume 4468 of *LNCS*. Springer, 2007.
30. 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.
31. H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *Proc. of FMOODS'02*. Kluwer.
32. 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.
33. 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.
34. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.