Model-Based Dependable Composition of Self-Adaptive Systems

Javier Cubo, Carlos Canal and Ernesto Pimentel
Dept. of Computer Science, University of Málaga, Campus de Teatinos, 29071, Málaga, Spain
Email: {cubo,canal,ernesto}@lcc.uma.es, Web: http://www.lcc.uma.es/~{cubo,canal,ernesto}

Keywords: self-adaptive, context-aware, SOA, transition systems, model transformation, dependable composition, adaptation, evolution, fault tolerance, error recovery

Received: October 15, 2010

Building mobile and pervasive systems as a selection, composition, adaptation and evolution of pre-existing software entities may arise dynamically and continuously different issues related to inconsistencies, changes or faults. We propose an approach to detect and handle these issues with the appropriate methodology in every case. This is performed by tackling three great challenges in software engineering related to self-adaptive systems: (i) their formalisation, by using model-based SOA, which bridge the business and software processes, (ii) their development and maintenance, by performing adaptation and/or evolution when inconsistencies or changes occur, and (iii) their monitoring to handle faults, by using error recovery techniques. We use an example based on an intelligent transportation system to validate our proposal.

Povzetek: Opisana je sestava prilagodljivih sistemov na osnovi modela.

1 Introduction

The increased usage of mobile and portable devices has given rise over the last few years to a new market of mobile and pervasive applications. These applications may be executed on either mobile computers (laptops, tabletPCs, etc.), or wireless hand-held devices (PDAs, smart phones, etc.), or embedded systems (PDAs, on-board computer, intelligent transportation or buildings systems, etc.), or even use sensors or RFID tags. Their main goal is to provide connectivity and services at any time, adapting and monitoring when required and improving the user experience. These systems are different to the traditional distributed computing systems. On one hand, a mobile system is able to change location allowing the communication via mobile devices. On the other hand, a pervasive application attempts to create an ambient intelligence environment to make the computing part of it and its enabling technologies essentially transparent. This results in some new issues related to inconsistencies, changes or faults, which arise dynamically and continuously whilst composing services in these systems, and which have to be detected and handled. These issues can be classified into four main categories [23]: (i) mismatch problems, (ii) requirement and configuration changes (iii) network and remote system failures, and (iv) internal service errors. The first refers to the problems that may appear at different interoperability levels (*i.e.*, signature, behavioural or protocol, quality of service and semantic or conceptual levels), and the Software Adaptation paradigm tackles these problems in a non-intrusive way [8].The second is prompted by continuous changes over time (new requirements or services fully created at run-time), and Software Evolution (or Software Maintenance) focuses on solving them in an intrusive-way [19]. The third and fourth are related to networks (network connection break-off or remote host unavailable) and services (suspension of services during the execution or system run-time error) failures respectively, that are both addressed by Fault Tolerance (or Error Recovery) mechanisms [24]. Developing real-world mobile and pervasive systems handling all these faults is extremely complex and error-prone. Therefore, it is essential to determine an effective methodology to develop this kind of system.

Self-adaptive software provides a broadly inclusive adaptation methodology that spans a wide range of adaptive behaviours [22]. One of the key aspects of self-adaptive software is that it supports both software adaptation and evolution, by addressing mismatch problems and requirement or configuration changes. Another advantage of self-adaptive systems is that they adapt the software systems to changing operational contexts and environments, thereby reducing human effort in the human-computer interaction. Context-awareness provides the most relevant information (location, identity, time and activity) to users and stakeholders, adapting themselves to their changing situation, preferences and requirements, and optimising the quality of service [12]. Therefore, context information plays an important role in software adaptation and evolution to control the scope of change. However, current programming technology offers only very weak support for developing context-aware applications, and new research is urgently needed to develop novel Context-Oriented Programming (COP) mechanisms [21]. As regards failures related to networks and services, fault tolerance mechanisms to be exploited for the development of dependable systems allow the handling of exceptions raised by adaptive demand, returning back the self-adaptive system to any earlier stable state. The choice of fault tolerance mechanisms depends on the fault assumptions and on the system's char-

acteristics and requirements. There are two main classes of error recovery [24]:backward and forward error recovery. The former is based on rolling services back to the previous correct state in the presence of failure. The latter involves transforming the system services into any correct state, and relies on an exception handling mechanism.

Self-adaptive systems requires high reusability, dependability, robustness, adaptability, and availability. In order to reduce efforts and costs, these systems may be developed using existing *Commercial-Off-The-Shelf* (COTS) components or (Web) services. In contrast to the traditional approach in which software systems are implemented from scratch, COTS and services can be developed by different vendors using different languages and different computer platforms. Although the reuse of software has matured and has overcome some of the previously mentioned problems, it has not become standard practice yet, since reusing components or services requires the selection, composition, adaptation and evolution of prefabricated software parts, by means of their public interfaces, in order to solve different problems. Thus, it is desired to reduce the effort of adapting and maintaining existing services in order to achieve a cost-effective and dependable development of self-adaptive systems. Component-Based Software Engineering (CBSE) [25] and Service-Oriented Architecture (SOA) [13] promote software reuse by selecting and assembling pre-existing software entities (COTS and services, respectively)[1]. These software development paradigms allow the building of fully working systems as efficiently as possible in order to improve the level of self-adaptive software reusability, dependability and adaptability [8]. On one hand, current industrial platforms using CBSE provide only some of the means to describe components at their signature level (*e.g.*, CORBA's IDL[2]). On the other hand, one way to implement SOA is using WSDL for describing services, SOAP[3] for communication, UDDI for service registry and discovery, and BPEL [1] for service orchestration. However, BPEL is not yet widely considered in current XML-based industrial service technology, which, in addition, only supports queries based on keywords and categories. This may bring about erroneous executions and/or low-precision results in realistic and complex applications, as it neither handles the order in which the service messages are exchanged with its environment, nor is it able to discover semantic capabilities of services (functionality) nor can it be adapted to a changing environment without human intervention. Therefore, behavioural descriptions, and multiple context and semantic (*e.g.*, by means of ontologies) information must be specified and managed in real-world services to avoid undesirable situations during their interaction, such as deadlocks or livelocks, and to improve their features (such as QoS). In this sense, our proposal tackles the need to support the variability of the adaptation process in self-adaptive systems by using context-aware, semantic-based, model-based adaptation, and dependency analysis mechanisms.

Approach and Contributions. We propose an approach to detect and handle the different inconsistencies, changes or faults arisen in self-adaptive systems. This is performed by tackling three great challenges in software engineering related to self-adaptive systems: (i) their formalisation, by using model-based SOA, which bridges the business and software processes, (ii) their development and maintenance, by performing adaptation and/or evolution when required, and (iii) their monitoring to handle faults, by using error recovery techniques.

In order to achieve these goals, we make the following contributions. Firstly (i) we develop a model transformation process to allow us the discovery, composition, adaptation and maintenance of services. This process, according to the Model-Driven Architecture (MDA)[4], takes a source model (BPEL or WF [10], both implemented as SOAs) and produces a target model (in our case transition systems), and vice versa. Secondly, (ii) we use software adaptation and evolution concern respectively with adapting or changing the software during its execution. Both paradigms typically tackle the adapting and the evolving of software separately depending on the changes being made. However, we propose a model-based approach using self-adaptive techniques through both paradigms to reduce the effort and cost of modifying the system. In this way, our approach will assist respectively to the application developers and software designers to first apply software adaptation (non-intrusive way) when that paradigm may solve the problem, and only in the case it is not enough, software evolution (intrusive way) will be used. Finally, (iii) we combine both backward and forward error recovery techniques to maintain consistency, correctness, and coordination of changes, and to handle errors in self-adaptive systems. We have developed a prototype tool on Python, which implements our approach, integrated inside the toolbox ITACA [7]. `ITACA`[5] (Integrated Toolbox for the Automatic Composition and Adaptation of Web Services) is a toolbox under implementation at the University of Málaga for the automatic composition and adaptation of services accessed through their interfaces. The toolbox fully covers an adaptation process which goes from behavioural model extraction from existing service interface descriptions, to the final adaptor implementation.

Figure 1 is an overview of our approach, which focuses on systems made up of a service repository, clients (considered as services as well), and a shared domain ontology. When a user performs a request, *e.g.*, from a mobile device, our process is executed. First, (1) abstract interface specifications (Context-Aware Symbolic Transition Systems, CA-STSs, presented in Section 3.1) are extracted from the BPEL or WF services, by means of our model transformation process (Section 3.2). Then, (2) a discov-

[1] In the sequel, we use service to refer both terms.

[2] www.omg.org/technology/documents/formal/components.htm

[3] http://www.w3.org/TR/soap/

[4] http://www.omg.org/docs/omg/03-06-01.pdf

[5] Accesible at http://itaca.gisum.uma.es

ery process based on semantic and compatibility mechanisms finds the services satisfying that request, and identifies possible mismatches and changes that will determine whether the services involved need adaptation and/or evolution (Section 4.1). If mismatches or changes occur, then (3) observation planning will determine *when*, *where*, *what*, and *how* [6] to perform adaptation and/or evolution depending on whether the changes are related to anticipated or unanticipated adaptation, respectively. Next, (4.a) in the case that adaptation is required, a CA-STS adaptor will be generated in a non-intrusive way (Section 4.1), and (4.b) if evolution is needed, then first the designer will have to modify the system in an intrusive way, and second the adaptor will be generated (Section 4.1). Then, (5) from the CA-STS adaptor, the corresponding BPEL or WF adaptor service is generated using our model transformation process (Section 4.1), and the whole system is deployed, allowing the BPEL or WF services to interact via the BPEL or WF adaptor. Finally, (6) a fault tolerance process handles exceptions raised by adaptive demand, returning back the system to any earlier stable situation, by using error recovery techniques (Section 4.2).

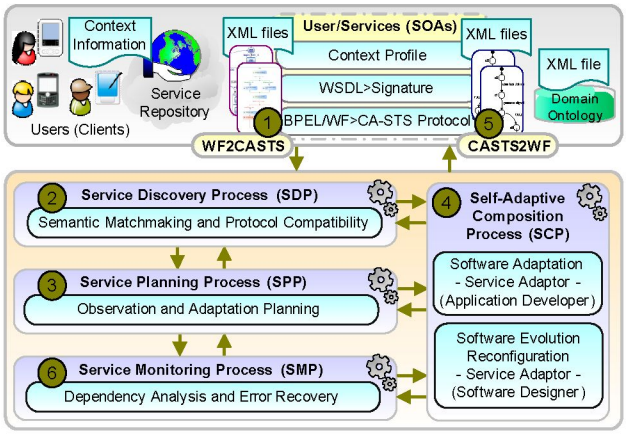

Figure 1: Overview of our proposal

Outline. The remainder of this article is structured as follows. In Section 2, we introduce a case study which will be used throughout this article for illustrative purposes. Section 3 presents our model-based SOA approach. In Section 4.1 the dependable composition process is described. Sections 5 presents works related to model-based transformation, and self-adaptive and error recovery techniques. Finally, Section 6 ends the article with a discussion about the evaluation of our approach and some concluding remarks.

2 Motivating example: ITS

To illustrate our proposal, we describe a case study in which services connected to an Intelligent Transportation System (ITS) require and provide context-aware transportation facilities. We consider different scenarios where city users of transport (passengers or drivers) are interested in planning their route on their hand-held devices (mobile phones or onboard computer), by receiving data from a Route service. In addition to the Route service, a Map service may also reply the user's requests as the Route service is not available anymore. We consider two kind of users: 1) bus/metro passenger, and 2) drivers. The latter have two different profiles: driving a private vehicle (car), or a taxi. Users receive different results of their route, depending on their profiles. We assume services can respond to the users' requests, but issues related to inconsistencies, changes or faults may arise at run-time, making it necessary to detect and handle them.

Passenger scenario. A passenger communicates with the system to obtain the best itinerary to a destination by bus/metro. The Route service response depends on certain context information, *i.e.*, the passenger location and destination, as well as the traffic or transport timetable, so the result may vary frequently.

Driver scenario. Let us imagine the Route service typically calculates a route requested by drivers based on traffic congestion (considering vehicles that enter and leave an area). In a normal situation, a car/taxi driver can change the route dynamically on being advised by the ITS of rerouting alternatives. In this scenario, we describe three different cases:

- A) Drivers request that the Route service considers the context information related to the weather as a new requirement to calculate the rerouting.

- B) From the previous case, vehicles driving in a specific area discover a new Car Parking service provided by the context of the new environment, but not considered initially by the system. Drivers would like to request this new service, so the ITS should include that service into the system.

- C) Considering the requirements of the two previous cases, we imagine that in a certain moment the connection with the Route service is lost. This service will be replaced automatically and quickly at run-time by another service with similar functions and considered at design-time by the system, *i.e.*, the Map service, which will also help to guide the driver.

This case study presents a service-oriented pervasive system with context-awareness features. Self-adaptive software, in addition to tackle different adaptive behaviours, is useful for dealing with all forms of embedded or pervasive software. We use a structured modelling approach to specify service-oriented architectures, because it is easier to determine when a new service is needed, as well as when it is more cost-effective and efficient to alter an existing service, develop a new one, or acquire a third-party service, and to manage fault tolerance mechanisms. Since models tend to be represented using a graphical notation, the model-based methodology involves using visual-modeling languages. We adopt an expressive and user-friendly graphical notation based on transition systems, which reduces the complexity of modelling services, as we will show in the next section.

3 Model-based SOA

In this section, we describe our formal model to specify services using Context-Aware Symbolic Transition Systems (CA-STS). Different automata-based or Petri net-based models can be used to describe behavioural interfaces. We have chosen CA-STS, which is based on transition systems, because it is simple, graphical, and provides a good level of abstraction to tackle discovery, verification, composition, or adaptation issues [14, 15]. Furthermore, any formalism to describe dynamic behaviour may be expressed in terms of a transition system [14]. Thus, our approach becomes general enough to be applied in other applications. In addition, we relate our interface model to implementation platforms. There exists several platforms or languages that can be used to develop services, such as UML[6], BPEL or WF. First, we present the syntax and operational semantics of our interface model. Second, we describe a textual grammar to abstract implementation details of WF activities, and define our transformation process to extract CA-STS specifications from WF services.

3.1 CA-STS Interface Model

We consider systems consisting of context-aware clients and services. We assume both client and service interfaces are specified using context profiles, signatures and protocols. Context profiles define information which may change according to the client preferences and service environment. Signatures correspond to operations profiles. Protocols are represented using transition systems. Client and services interact according to the operational semantics we will define later.

Context Profile, Signature and Protocol.
A *context* is defined as *"the information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to interaction between a user and an application including the user and application themselves"* [12]. Context information can be represented in different ways and can be classified in four main categories [17]: (i) user context: role, preferences, language, calendar, social situation or privileges, (ii) device/computing context: network connectivity, device capabilities or server load, (iii) time context: current time, day, year, month or season, and (iv) physical context: location, weather or temperature. For our purpose, we only need a simple representation where contexts in both clients and services are defined by *context attributes* with associated values. In addition, we differentiate between static context attributes (*e.g.*, role, day, ...) and dynamic ones (*e.g.*, network connectivity, current time, location, ...). Dynamic attributes can change continuously at run-time, so they have to be dynamically evaluated during the service composition. Finally, both clients and services are characterised by public (*e.g.*, weather, temperature, season, ...) and private (*e.g.*, personal data, local resources, ...) context attributes. Thus, we represent and gather the service context information by using a *context profile*, which is a set of tuples (CA, CV, CK, CT), where: CA is a context attribute (or simply context) with its corresponding value CV, CK determines if CA is static or dynamic, and CT indicates if CA is public or private. For instance, (*user, driver, static, public*), indicates that *user* is a public and static context which corresponds to the user profile *driver* as value.

A *signature* corresponds to a set of operation profiles. This set is a disjoint union of provided and required operations. An operation profile is the name of an operation, together with its argument types (input/output parameters) and its return type.

A *protocol* is represented using a Symbolic Transition Graph (STG) [16] extended with value passing, context variables and conditions, that we call Context-Aware Symbolic Transition System (CA-STS). Conditions specify how applications should react (*e.g.*, to context changes). We take advantage of using ontologies described in a specific domain to capture and manage the semantic information of the services in a system by comparing concepts, such as context information, operation names, arguments and types. In this way, we can determine the relationship between the different concepts that belong to that domain.

Let us introduce the notion of variable, expression, and label required by our CA-STS protocol. We consider two kinds of *variables*, those representing regular variables or static context attributes, and those corresponding to dynamic context attributes (named context variables). In order to distinguish between them, we will mark the context variables with the symbol "$\sim$" over the specific variable. An *expression* is defined as a variable or a term constructed with a function symbol f (an identifier) applied to a sequence of expressions, $i \in f(F_1, \ldots, F_n)$, F_i being expressions.

Definition 1 (CA-STS label). *A label corresponding to a transition of a CA-STS is either an internal action τ (tau) or a tuple (B, M, D, F) representing an event, where: B is a condition (boolean expression that manages both conditional choices and context changes), M is the operation name, D is the direction of operations (! and ? represent emission and reception, respectively), and F is a list of expressions if the operation corresponds to an emission, or a list of variables if the operation is a reception.*

Definition 2 (CA-STS Protocol). *A Context-Aware Symbolic Transition System (CA-STS) Protocol is a tuple (A, S, I, Fc, T), where: A is an alphabet which corresponds to the set of CA-STS labels associated to transitions, S is a set of states, $I \in S$ is the initial state, $Fc \subseteq S$ are correct final states (deadlock-free), and $T \subseteq S \times A \times S$ is the transition function whose elements $(s_1, a, s_2) \in T$ are usually denoted by $s_1 \xrightarrow{a} s_2$.*

Finally, a *CA-STS interface* is constituted by a tuple (CP, SI, P), where: CP is a context profile, and SI is the signature of the CA-STS protocol P. Both client and services consist of a set of interfaces. For instance, let us

[6] http://www.omg.org/technology/documents/formal/uml.htm

focus on the client shown in Figure 6. It has an interface (CP_U, SI_U, P_U), where CP_U refers to the context information related to the user location (dynamic context attribute *loc*), user profile and device used by the client of the user (static context attributes *user* and *dev* respectively), SI_U is formed by all the operation profiles, such as $l_{u_1} = reqR!dest, \tilde{loc}, us\tilde{er}$, and P_U is the protocol which indicates the CA-STS behaviour. For example, l_{u_1} means that a client with the context information *loc* and *user* issues an emission looking for a route from his/her location to a destination, and then this client receives a possible route $l_{u_2} = getR?route$, and so on. Note we have left out the return types of the arguments to simplify the notation. Initial and final states are depicted in CA-STSs using bullet arrows and hollow states, respectively. Our proposal is suitable for synchronous systems where clients interact with services, such as mobile systems. We adopt a synchronous and binary communication model (explained in next section, Figure 3). Clients can execute several protocols simultaneously, *i.e.*, concurrent interactions (in a binary model). Client and service protocols can be instantiated several times.

At the user level, client and service interfaces can be specified by using: (i) context information into XML files for context profiles, (ii) WSDL for signatures, and (iii) business processes defined in industrial platforms, such as Abstract BPEL (ABPEL) [1] or WF workflows (AWF) [10], for protocols. We assume context information is inferred from the client requests (HTTP header of SOAP messages), thereby as a change occurs the new value of the context attribute is automatically sent to the corresponding service (controlled in rules presented in Figure 2). We also consider processes (clients and services) implemented as business processes which provide the WSDL and protocol descriptions.

Next, we define the CA-STS operational semantics.

Operational Semantics of CA-STS.

We formalise first the operational semantics for one CA-STS service, and second for the composition of n CA-STS services. In the following, we use a pair $\langle s, E\rangle$ to represent an active state $s \in S$ and an environment E. An environment is a set of pairs $\langle x, v\rangle$ where x is a variable, and v is the corresponding value of x (it can be also represented by $E(x)$). The function *type* returns the type of a variable. We use boolean expressions b to describe CA-STS conditions. Regular and context variables are evaluated in emissions and receptions (by considering the current value of the context, *e.g.*, the current date), respectively. Therefore, two evaluation functions are used to compute expressions in an environment: (i) ev evaluates regular variables or expressions, and (ii) ev_c evaluates context variables changing dynamically. We define ev as follows:

$$ev(E,x) \triangleq \begin{cases} E(x) & \text{if } x \text{ is a regular variable} \\ x & \text{if } x \text{ is a context variable} \end{cases}$$

$$ev(E, f(v_1,\ldots,v_n)) \triangleq f(ev(E,v_1),\ldots,ev(E,v_n))$$

Function ev_c is defined in a similar way to ev, only considering context variables, since we first apply ev to evaluate the regular variables: $ev_c(E,x) \triangleq E(x)$, where x is a context variable. We also define an environment overloading operation "$\oslash$" in such a way that given an environment E, $E \oslash \langle x, v\rangle$ denotes a new environment, where the value corresponding to x is v.

We present in Figure 2 the semantics of a CA-STS ($\rightarrow_o$), with three rules that formalise the meaning of each kind of CA-STS label: internal actions τ (INT), emissions (EM), and receptions (REC); and one rule to simulate the dynamic update of the environment according to the context changes at run-time (DYN). Note that according to Definition 1, $b \in B$ is a condition, $a \in M$ is an operation name, and $x \in F$ and $v \in F$ correspond to a list of variables and expressions, respectively. A condition b may contain regular and/or context variables and both of them must be evaluated in the environment of the source service (sender), because the decision is taken in the sender. However, evaluation of expressions v only affects regular variables (rule EM), since context variables will be evaluated in the target service (receiver) to consider the context values when the message is received (see rule COM in Figure 3). We assume that the dynamic modification of the environment will be determined by different external elements depending on the type of the context (*e.g.*, user intervention, location update by means of a GPS, time or temperature update, and so on). Then, we model this situation by assuming a transition relation which indicates the environment update as a change occurs, denoted by $E \rightsquigarrow_d E'$, where $E'(x) \neq E(x)$ only if x is a dynamic context variable, and in which case the new value of x is automatically sent to the corresponding service.

$$\frac{(s \xrightarrow{b,\tau} s') \in T \quad ev_c(ev(E,b),b) = \texttt{true}}{\langle s,E\rangle \xrightarrow{\tau}_o \langle s',E\rangle} \quad \text{(INT)}$$

$$\frac{(s \xrightarrow{b,a?x} s') \in T \quad ev_c(ev(E,b),b) = \texttt{true}}{\langle s,E\rangle \xrightarrow{a?x}_o \langle s',E\rangle} \quad \text{(REC)}$$

$$\frac{(s \xrightarrow{b,a!v} s') \in T \quad ev_c(ev(E,b),b) = \texttt{true} \quad v' = ev(E,v)}{\langle s,E\rangle \xrightarrow{a!v'}_o \langle s',E\rangle} \quad \text{(EM)}$$

$$\frac{E \rightsquigarrow_d E'}{\langle s,E\rangle \xrightarrow{\tau}_o \langle s,E'\rangle} \quad \text{(DYN)}$$

Figure 2: Operational Semantics of one CA-STS

The operational semantics of n ($n > 1$) CA-STSs ($\rightarrow_c$) is formalised using two rules: a first synchronous communication rule (COM, Figure 3) in which value-passing and variable substitutions rely on a late binding semantics [20] and where the environment E is updated; and a second independent evolution rule (INE$_\tau$, Figure 3). A list of pairs $\langle s_i, E_i\rangle$ is represented by $[as_1,\ldots,as_n]$. Rule COM uses the function ev_c to evaluate dynamically in the receiver the context changes related to the dynamic context attributes of the sender. Regular variables have been evaluated previously in the rule EM when the message is emitted. This dynamic evaluation handled in the operational semantics allows the modelling of service protocols depending on con-

text changes. Rule INE_τ is executed in the case of an internal service propagation that gives rise to either a state (related to the rule INT) or an environment (rule DYN) change. Thus, transitions $\rightarrow_c$ do not distinguish between internal evolutions coming from either internal actions in services or dynamic updates in the environment.

$$\frac{i,j \in \{1..n} \quad i \neq j \quad type(x) = type(v) \quad \langle s_i, E_i\rangle \xrightarrow{a!v}_o \langle s_i', E_i\rangle \quad \langle s_j, E_j\rangle \xrightarrow{a?x}_o \langle s_j', E_j\rangle \quad E_j' = E_j \oslash \langle x, ev_c(E_j, v)\rangle}{[as_1, \ldots, \langle s_i, E_i\rangle, \ldots, \langle s_j, E_j\rangle, \ldots, as_n] \xrightarrow{a!v}_c [as_1, \ldots, \langle s_i', E_i\rangle, \ldots, \langle s_j', E_j'\rangle, \ldots, as_n]} \quad \text{(COM)}$$

$$\frac{i \in \{1..n} \quad \langle s_i, E_i\rangle \xrightarrow{\tau}_o \langle s_i', E_i'\rangle}{[as_1, \ldots, \langle s_i, E_i\rangle, \ldots, as_n] \xrightarrow{\tau}_c [as_1, \ldots, \langle s_i', E_i'\rangle, \ldots, as_n]} \quad (\text{INE}_\tau)$$

Figure 3: Operational Semantics of n CA-STSs

Following, we present our model transformation process by using WF services as illustration purpose.

3.2 Model Transformation Process

To perform the service discovery, composition, adaptation and maintenance, we first need to define a textual notation to abstract and formalise services implemented in the WF platform. Second, we define our model transformation process.

Abstraction of WF Workflows.
To relate our model transformation process with realistic and complex examples, we use the WF platform, which belongs to the .NET Framework 3.5 and is supported by Visual Studio 2008. We have chosen WF because it makes the implementation of services easier thanks to its workflow-based graphical support and the automation of the code generation, and it is an useful and interesting alternative compared to the well-know BPEL. Nevertheless, we have also validated our proposal using BPEL as shown in [7]. In addition, the .NET Framework is widely used in many companies, and WF is increasingly prevalent in the software engineering community [26].

In order to illustrate the motivating example presented in Section 2, we use a representative kernel of the WF activities, namely `Code`, `Sequence`, `Terminate`, `Receive`, `Send`, `IfElse`, `While`, and `Listen` with `EventDriven` activities, that are general enough to describe any service.

In Table 1, we formalise the textual grammar (left hand-side) defined for the WF activities considered (on the right hand-side the informal meaning of these activities is provided), which abstracts several implementation details. Our grammar considers as input textual workflows (defined in XML files) corresponding to the graphical description of the WF workflows, with WF activities $\mathscr{A}$, where C, C_i are boolean conditions, I, I_i (inputs), O, O_i (outputs) are parameters of activities, and Id are service identifiers.

The WF platform is capable of developing workflows in different scenarios, from simple sequential ones to realistic and complex state machine-based workflows involving human interaction. The programming languages available in the platform are *Visual Basic* and *C#*. Our examples have been implemented in *C#*.

$\mathscr{A}$ ::= `Code`	*executes code*
| `Terminate`	*ends WF*
| `Receive(`$Id,Op[,O,I_1,\ldots,I_n]$`)`	*receives msg*
| `Send(`$Id,Op[,O_1,\ldots,O_n,I]$`)`	*sends msg*
| `Sequence(`$\mathscr{A}_1,\mathscr{A}_2$`)`	*executes* $\mathscr{A}_1,\mathscr{A}_2$
| `IfElse(`$(C_1,\mathscr{A}_1),\ldots,(C_n,\mathscr{A}_n),\mathscr{A}$`)`	$\mathscr{A}_i$ *if* C_i *or* $\mathscr{A}$
| `While(`$C,\mathscr{A}$`)`	$\mathscr{A}$ *while* C
| `Listen(`$E_1,\ldots,E_n$`)`	*fires one* E_i
E ::= `EventDriven(Receive(`$Id,Op[,I_i]$`),`$\mathscr{A}$`)`	$\mathscr{A}$ *when* Id

Table 1: Grammar for the WF abstract notation

Example. We have designed WF workflows for the User Route request, and for the Route and Map services. WF provides a WSDL description for each WF workflow. For space reasons, in Figure 4 only the WF workflow that represents the behaviour of the User Route request is shown.

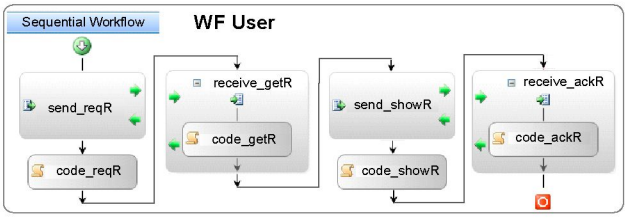

Figure 4: WF workflow of the User's request

Next, we present how we extract CA-STS specifications from WF services.

From WF to CA-STSs.
CA-STSs are used as an abstraction to focus on behavioural composition issues by describing service interfaces in a standard notation. These CA-STSs are automatically generated from WF services. For each WF service, our model transformation process parses the three XML files corresponding to its context information, WSDL description, and WF workflow. A new XML file containing the information about its context profile, signature, and CA-STS protocol is automatically generated. This XML corresponds to the behavioural interface of a CA-STS specification. This process has been implemented following the patterns of our transformation process presented in Figure 5.

We have developed an ad-hoc transformation language to translate WF activities (WF workflows defined in XML files) in CA-STS elements (XML files represented in a graphical notation by means of transition systems) and vice versa. The extracted CA-STS specifications must preserve the semantics of workflows as encoded in the WF platform. A formal proof of semantics preservation between both levels has not been achieved yet since the WF formal semantics is not rigorously documented. Our encoding has been deduced from our experiments using the WF platform. The main ideas of the CA-STS specification obtained from abstract description of workflow constructs are the following: (i) `Code` is an internal transition, (ii) `Terminate` corresponds to a final state, (iii) `Receive` and `Send` are reception and emission, respectively, (iv) `Sequence` must

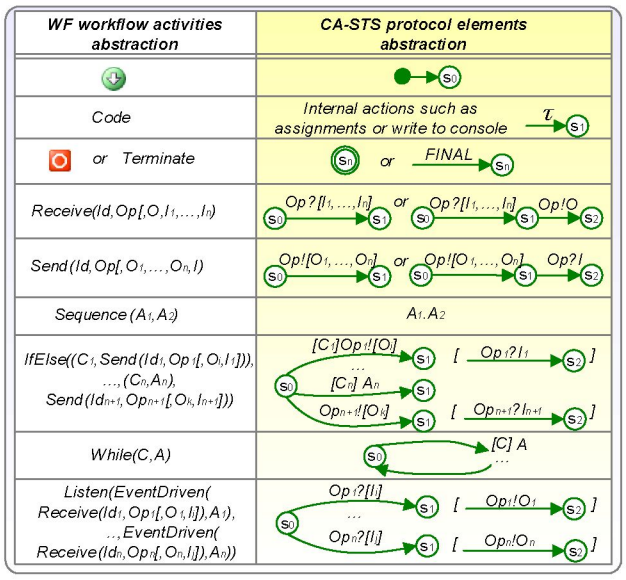

Figure 5: Patterns of our model transformation process from WF to CA-STS and vice versa

preserve the activities' order, (v) `IfElse` corresponds to an internal choice, (vi) `While` is translated as a looping behaviour, and `Listen` corresponds to an external choice. Initial and final states in the CA-STS come respectively from the initial and final states that appear in the workflow. There is a single initial state that corresponds to the beginning of the workflow. Final states correspond either to a `Terminate` or to the end of the workflow, so several final states may appear in the CA-STS because several branches in the workflow may lead to a final state.

Example. We apply the model transformation process to the WF services of our case study in order to obtain the corresponding CA-STS specifications. Figure 6 shows the interfaces of the User (passenger or driver) and the Route and Map services modelled using our CA-STS interface model. Each interface has a context profile, a signature and a CA-STS protocol.

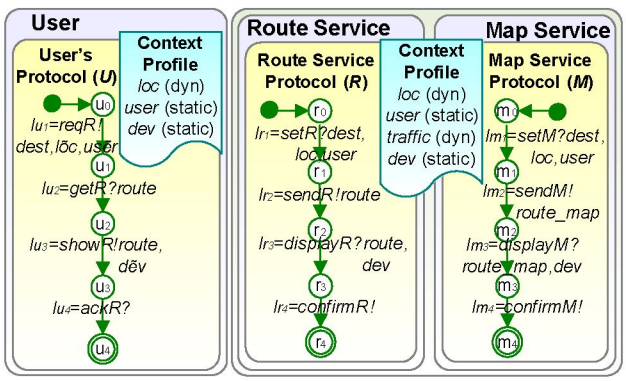

Figure 6: CA-STS of User and Route and Map services

4 Dependable composition of self-adaptive SOA

This section presents our approach to tackle self-adaptive systems changing dynamically over time and must continue offering services as inconsistencies, changes or faults occur. We aim to combine self-adaptive composition and error recovery techniques to perform adaptation and evolution strategies and to handle errors, respectively.

4.1 Self-Adaptive Composition

Composing services relates to dealing with assembly of autonomous services given their interfaces. We need to address the specification of the composition, and to ensure the services are composed in a consistent way.

Firstly, our discovery process (SDP module in Figure 1) finds the most appropriate services for a user's request. To do that, it is based on semantic matchmaking and protocol compatibility techniques [9]. The first is used to establish a ranked list of the services that better match the user's request, by comparing the semantic matching of the context profiles and all the operation profiles (names, arguments and types) *w.r.t.* an ontology defined in the ITS domain. The second checks if the services selected are compatible with the user at the protocol level. There exists different notions of compatibility in synchronous communication, such as opposite behaviours, unspecified reception, and deadlock-freeness [4]. We have chosen the deadlock-freeness notion to illustrate our proposal, but other definitions could also be used. This compatibility definition guarantees that all the interactions between two services are performed in a satisfactory way, leading to a correct final state.

Secondly, once our approach discovers services, changes during the service composition may occur in many different ways. On one hand, when adaptability is anticipated and limited to some variation points (*e.g.*, software product line), the different changes to be adapted at run-time are known at design-time. On the other hand, in the unanticipated adaption, the possible variations are recognised and computed at run-time, being, for instance, new services discovered and assembled dynamically using self-awareness and environmental context information by means of planning techniques. Planning (SPP module in Figure 1) is a key feature for self-adaptive systems. Observation planning determines *when*, *where*, *what*, and *how* [6] to perform adaptation and/or evolution to solve faults. Adaptation planning aims to prepare the system to be adapted by using an *adaptation contract*. Software adaptation covers all the changes related to the anticipated adaptation. In addition, it is also characterised by highly dynamic run-time procedures that occur as devices and applications move from network to network, changing their contexts, and enhancing the flexibility and maintainability of systems. Therefore, software adaptation can also address these cases of unanticipated adaptation. Software evolution refers to the continuous changes over time, tackling other cases of unan-

ticipated adaptation, such as new requirements or services fully created at run-time.

Then, self-adaptive techniques by combining both adaptation and evolution paradigms (SCP module in Figure 1) supposes a contribution of the approach presented in this work, where the actors are the application developers and the software designers, respectively. We have made the distinction between software adaptation, where application developers generate third-party adaptors (using the adaptation contract) in a non-intrusive way, and software evolution, where software designers modify the software entities in an intrusive way and then an adaptor is generated. To perform this, we follow a two-process methodology, by modeling self-adaptive systems with a combination of anticipated and unanticipated adaptation. First, the application developers, who do not have knowledge of the source code and documentation, take advantage of adaptors to automatically adapt software when it is not necessary to modify the code. When the first process is not enough to adapt the system to the new situation because changes in the requirements or an addition/removal of a service occur, then our approach help the software designers to perform evolution. Therefore, they select a minimal set of changes to adapt software, as they are familiar with such software system. Note that this intrusive way of adapting the system requires that the designer has knowledge not only about the system, but also about our approach.

From the (matching) tuples of sets of correspondences obtained in the discovery process, we can automatically generate an adaptation contract when any fault or change occurs during the service interaction. Moreover, we also want composition to distinguish between the available contexts when translating the messages among services. Using a non-contextual approach, message correspondences are fixed. This prevents inconsistencies or changes in these connections being taken into account, and motivates the need for the new capabilities that our approach provides in order to achieve message translation depending on contexts. Therefore, we define the adaptation contract between events in the CA-STS protocols by means of vectors expressing interactions among service messages to specify the evolution of every service depending on its contexts. These interactions denote a service communication and are formalised through *synchronisation vectors* [2], which allow messages with different names and even different numbers of parameters to be synchronised. Each event appearing in one vector is executed by one service, and the overall result corresponds to a synchronisation between all the services involved. A vector may involve any number of services.

Definition 3 (Synchronisation Vector). *A synchronisation vector (or vector for short) for a set of protocols* $P_i = (A_i, S_i, I_i, Fc_i, T_i)$, $i \in \{1,..,n\}$, *is a tuple* $\langle v_1, \ldots, v_n \rangle$ *with* $v_i \in A_i \cup \{\varepsilon\}$, ε *meaning that a service does not participate in a synchronisation.*

However, vectors are not sufficient to support more advanced adaptation scenarios such as contextual rules, choice between vectors or, more generally, ordering (*e.g.*, when one message in some service corresponds to several in another service, which requires the application of several vectors). The order in which vectors have to be applied can be specified using different notations such as regular expressions, Labelled Transition Systems (LTSs), or (Hierarchical) Message Sequence Charts (MSCs). Due to their readability and user friendliness, we chose to specify adaptation contracts using LTSs whose labels are tuples. This tuple-LTS is made up of a set of tuples $\langle v, a \rangle$, where v is a vector on transitions and a indicates if v has been executed, interrupted or not executed (values can be `successful_execution`, `int_execution` or `not_executed` represented with S, I and N, respectively). Therefore, this tuple-LTS is essential in some situations in which faults, such as deadlocks or livelocks, can be avoided by applying some vectors in a specific order. If the order among correspondences between services does not matter, the tuple-LTS contains one state with all transitions looping on it.

Next, we introduce the formal notion of adaptation contract, which is used to model the composition of services making use of vectors and tuple-LTS.

Definition 4 (Adaptation Contract). *An adaptation contract for a set of services* Ws_i, $i \in 1,...,n$, *is defined as a couple* (V_{Ws_i}, T_{lts}), *where* V_{Ws_i} *is a set of vectors for services* Ws_i, *and* T_{lts} *is a tuple-LTS that indicates the interaction order of the vectors* V_{Ws_i}.

Finally, by using the adaptation contract and CA-STS services, we generate a third-party CA-STS adaptor, that is in charge of coordinating the services in the system *w.r.t.* the set of interactions defined in the contract (according to the rule COM, Figure 3). For limitations of space and since it is not a new contribution of this article, the adaptor generation is detailed in our previous works [7, 10]. Adaptor is platform independent, and it can be refined *w.r.t.* a specific platform, such as the WF platform (using our transformation process, Figure 5).

Next, we describe our fault tolerance process, which handles exceptions by using error recovery technique.

4.2 Error Recovery Mechanism

Monitoring (SMP module in Figure 1) is necessary to maintain consistency, correctness, and coordination of changes, as well as to handle errors. We focus on atomic actions, that allow programmers to apply both backward and forward error recovery. These techniques use appropriate exception handling mechanisms, which enable dealing with dependability of composed services. Exception handling is the method of building a system to detect and recover from exceptional conditions (unexpected occurrences). Protecting a system from the effects of exceptional conditions is a difficult task, since all unexpected occurrences can not be anticipated easily while designing the system. It is necessary to build exception handlers in order to detect and handle these exception conditions by avoiding application failures. We perform fault tolerance

mechanisms to handle exceptions raised by adaptive demand, returning back the system to any earlier stable state. To do that, we define an *error recovery algorithm* based on backward and forward error recovery, handling possible failures.

First, we need to define a data structure, called *vector dependency*, to track dependencies among the synchronisation vectors of an adaptation contract. We base this on the tuple-LTS previously generated to obtain the vector dependencies, since the tuple-LTS indicates the order of interactions of the vectors.

Definition 5 (Vector dependency). *A vector dependency between two vectors v_1 and v_2 is a link relationship such as $v_1 \xrightarrow{P} v_2$, where P can define both either functional or non-functional properties (such as temporal requirements or resources), and it must always be true to move from v_1 to v_2.*

Then, we define an *interaction set*, which is generated as a set of vector dependencies. This set is used to handle failures, by identifying all the vectors affected by these failures. Thus, an interaction set contains all the vectors in the adaptation contract of the communication between services involved in the interaction.

Our algorithm is executed when any failure related to networks or services occurs, by performing the following steps: (1) identify the last vector to be executed in the interaction set where the fail occurred, (2) change the status of all vectors of that interaction set whose events are directly involved in the error to `int_execution`, (3) change the status of all vectors related to other interaction sets which depended on the vectors involved in the failed interaction set to `int_execution`, (4) wait for a timeout if the service that provoked the error can be re-established, or swap the failed service with another service capable of performing similar roles, (5) if there are not services to swap, then an exception will be triggered to all the vectors involved in the error and the execution will stop, otherwise (6) re-execute all vectors in the interaction set that are labeled as `int_execution` and therefore change the value of those vectors to `successful_execution`.

Next, we illustrate both self-adaptive and error recovery processes by using the different scenarios described in our case study.

Example. Considering the full approach presented above, we address the scenarios of our case study ITS.

Firstly, common to all the scenarios, to illustrate the discovery process, we focus on the user's request (passenger or driver). Our process selects Route and Map services in that order according to the semantic matchmaking, and two (matching) tuples of sets of correspondences between operation profiles are returned and presented below (labels l_{u_1}, l_{r_1}, etc., are represented in Figure 6).

$MT_{U,R} = \{(l_{u_1}, l_{r_1}), (l_{u_2}, l_{r_2}), (l_{u_3}, l_{r_3}), (l_{u_4}, l_{r_4})\}$

$MT_{U,M} = \{(l_{u_1}, l_{m_1}), (l_{u_2}, l_{m_2}), (l_{u_3}, l_{m_3}), (l_{u_4}, l_{m_4})\}$

Once our process has discovered the services, we need to handle the inconsistencies, changes or faults which have arisen while composing services in our four scenarios.

Passenger scenario. In this scenario, our self-adaptive process applies software adaptation due to the mismatch problems in the behavioural interfaces. An adaptor is generated by means of the adaptation contract between the User (passenger) and the Route service. The contract is made up of the set of vectors presented below and the tuple-LTS depicted in Figure 7. The ITS knows at design-time the different contexts considered at run-time in this scenario, so it is enough with anticipated adaptation for the response given by the Route service.

$\{v_1 = \langle l_{u_1}, l_{r_1}\rangle, v_2 = \langle l_{u_2}, l_{r_2}\rangle,$
$v_3 = \langle l_{u_3}, l_{r_3}\rangle, v_4 = \langle l_{u_4}, l_{r_4}\rangle\}$

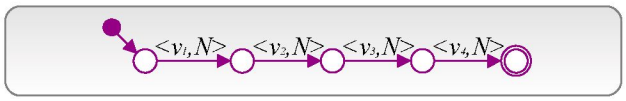

Figure 7: Tuple-LTS indicating the interaction order between the User and the Route service

It is worth mentioning that in every tuple $\langle v_i, a\rangle$, a is always initialised to N (`not_executed`), and during the composition process this value will change to either S (`successful_execution`) when the vector v_i is executed or to I when it is interrupted (`int_execution`).

Driver scenario. Here we have three different cases.

- A) and **- B)** Both cases needs unanticipated adaptation based on software evolution. Neither the new requirement (the context information related to weather) requested by drivers to obtain the rerouting, nor the new service (Parking service) provided at run-time by the driver location, were considered by the ITS at design-time. Therefore, the software designer has to modify the code of the Route service to include the weather in the context profile, and to incorporate the new Car Parking service into the ITS. Our approach will reconfigure dynamically the new full system to allow to the Users (drivers) to carry on communicating correctly with the Route service, and to discover the new service when they require it. The new CA-STS interfaces corresponding to the User and the Route, Map and Parking services are shown in Figure 8.

Note that the modifications are represented by dashed lines and bold text (*e.g.*, weather). In addition, conditions have been added (*e.g.*, $[user == \text{“}passenger\text{”}]$ in l_{u_3}) to determine that only user profile driver will request the parking service. The designer needs to know about the system and our approach to perform these kinds of modifications.

We continue focusing on the interaction between the User and Route service, but now also including the Parking service. Then, software adaptation is applied in the new service interfaces, by generating a new adaptation contract to avoid the new mismatches. Below we present the set of vectors (labels l_{u_1}, l_{r_1}, etc., are represented in Figure 8), and the tuple-LTS (Figure 9).

$\{v_1 = \langle l_{u_1}, l_{r_1}\rangle, v_2 = \langle l_{u_2}, l_{r_2}\rangle, v_3 = \langle l_{u_3}, l_{r_3}\rangle,$
$v_4 = \langle l_{u_4}, l_{r_4}\rangle, v_5 = \langle l_{u_5}, l_{p_1}\rangle, v_6 = \langle l_{u_6}, l_{p_2}\rangle,$
$v_7 = \langle l_{u_7}, l_{p_3}\rangle, v_8 = \langle l_{u_8}, l_{p_4}\rangle, v_9 = \langle l_{u_9}, l_{r_3}\rangle,$
$v_{10} = \langle l_{u_{10}}, l_{p_5}\rangle\}$

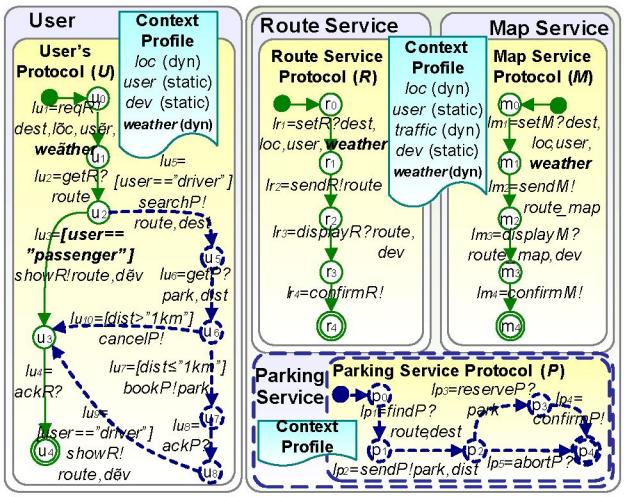

Figure 8: CA-STS of User and Route, Map and Parking services after applying our process in A) and B)

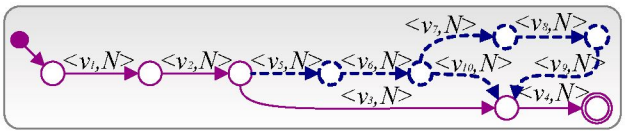

Figure 9: Tuple-LTS indicating the interaction order between the User and the Route and Parking services

Finally, we can generate an adaptor for the interaction between the User and Route and Parking services.

- C) This case considers the modifications performed in the previous cases. Our error recovery algorithm is applied here, since a suspension of the Route service during the service interaction occurred.

Before performing our algorithm, we generate the vector dependencies and the interaction sets corresponding to the communication between the User and the Route and Parking services, by means of the tuple-LTS presented in Figure 9.

$$\{v_1 \xrightarrow{P_{v_1,v_2}} v_2,\ v_2 \xrightarrow{P_{v_2,v_3}} v_3,\ v_2 \xrightarrow{P_{v_2,v_5}} v_5,$$
$$v_3 \xrightarrow{P_{v_3,v_4}} v_4,\ v_5 \xrightarrow{P_{v_5,v_6}} v_6,\ v_6 \xrightarrow{P_{v_6,v_7}} v_7,$$
$$v_6 \xrightarrow{P_{v_6,v_{10}}} v_{10},\ v_7 \xrightarrow{P_{v_7,v_8}} v_8,\ v_8 \xrightarrow{P_{v_8,v_9}} v_9,$$
$$v_9 \xrightarrow{P_{v_9,v_4}} v_4,\ v_{10} \xrightarrow{P_{v_{10},v_4}} v_4\}$$

We assume properties P_{v_i,v_j} are defined correctly according to the requirements of the user's request. The interaction sets are as follows: $I_{u,r} = \{v_1, v_2, v_3, v_4, v_9\}$ and $I_{u,p} = \{v_5, v_6, v_7, v_8, v_{10}\}$, corresponding to the communication of the User with the Route service and the User with the Parking service, respectively.

Now, to illustrate the algorithm, we assume that during the interaction between the User and the Route and Parking services, a failure occurs in vector v_4 (of the previous contract) corresponding to the confirmation of the Route service, *i.e.*, the correspondence between $l_{u_4} = ackR?$ and $l_{r_4} = confirmR?$). Therefore, our process changes to `int_execution` all the vectors involved in that error, and automatically selects another service previously considered and discovered, *i.e.*, the Map service, which replaces the Route service at run-time. This is possible because the designer developed the ITS to support a possible connection loss of the Route service, so a reconfiguration of the system is unnecessary, which reduces effort and cost.

A new adaptation contract (vectors and tuple-LTS), with its corresponding adaptor, is generated to solve the new mismatch problems in the interaction of the User and the Map and Parking services.

$\{v_1 = \langle l_{u_1}, l_{m_1}\rangle, v_2 = \langle l_{u_2}, l_{m_2}\rangle, v_3 = \langle l_{u_3}, l_{m_3}\rangle,$
$v_4 = \langle l_{u_4}, l_{m_4}\rangle, v_5 = \langle l_{u_5}, l_{p_1}\rangle, v_6 = \langle l_{u_6}, l_{p_2}\rangle,$
$v_7 = \langle l_{u_7}, l_{p_3}\rangle, v_8 = \langle l_{u_8}, l_{p_4}\rangle, v_9 = \langle l_{u_9}, l_{m_3}\rangle,$
$v_{10} = \langle l_{u_{10}}, l_{p_5}\rangle\}$

The corresponding tuple-LTS is equivalent to that presented in Figure 9, but replacing the synchronisation vectors v_1, v_2, v_3, v_4, v_9 of the previous contract (with Route service) with the vectors v_1, v_2, v_3, v_4, v_9 related to the new contract (with Map service).

5 Related work

This section compares our approach with related works in software composition, especially those which focus on model-based transformation, software adaptation and/or evolution, and error recovery.

With respect to the relationship between existing programming languages and platforms, the work presented in [5] outlines a methodology for the automated generation of adaptors capable of solving behavioural mismatches between BPEL processes (some interaction scenarios cannot be resolved). In [3], the authors present techniques to provide semi-automated support for identification and resolution of mismatches between service interfaces and protocols, and generate adaptation behavioural specifications based on SCA architecture. Compared to these works, we generate WF adaptor services that consider not only signature and protocol mismatches, but also context-aware and semantic issues. In addition, our approach is able to reorder messages among services when required, since our discovery process allows this facility automatically. This is necessary to ensure correct interaction in the case where communicating entities have messages which are not ordered as required.

Some research works have tackled software adaptation and evolution in an architecture-driven style [22], or repair programs by means of recommending adaptive changes [11]. Another important dimension is using formal methods to describe software systems more formally and to understand the cause of changes (domain structure) [8, 21]. In [19], several approaches for supporting static or dynamic adaptability and evolvability by means of a wide diversity of research domains (requirements, architecture, data, runtime and language evolution, SOAs), are presented. In our approach, we take advantage of both adaptation and/or evolution, in a model-based approach, depending on the needs of anticipated and unanticipated adaptation, and the four

categorising features, *when*, *where*, *what*, and *how*.

As regards fault tolerance mechanisms, one of the most beneficial ways of applying fault tolerance is by associating its measures with system structuring units [24]. Structuring units, which decrease system complexity and make it easier for developers to apply fault tolerance, can be: distributed transactions and atomic actions. Distributed transactions use backward error recovery [18] as the main fault tolerance measure in order to satisfy the ACID (atomicity, consistency, isolation, durability) properties. Transactions suppose a powerful abstraction to address failures occurring in closed systems. However, they impose highly severe constraints over systems in open environments such as SOA (*e.g.*, real-time systems do not have time to go back). In our approach, we use atomic actions, that allow programmers to apply both backward and forward error recovery to satisfy certain properties for composing service as a failure occur. Forward error recovery uses appropriate exception handling without impacting on the autonomy of services whilst exploiting their possible support for dependability. In addition, to handle exceptions optimally, our error recovery mechanism specifies that services return exceptions quickly, since notification delay can affect the SOA performance, especially in complex workflow systems.

Summarising, our approach combines efforts to detect and handle the different changes or faults arising in self-adaptive systems, by modelling SOA, performing adaptation and/or evolution when required, and monitoring failures with error recovery techniques.

6 Discussion and conclusions

Self-adaptive software requires high dependability, robustness, adaptability, and availability. Our approach maintains system consistency and integrity by examining each change and removing those that render the system inconsistent or unsafe. We focus on the development and maintenance of reliable software systems through self-adaptive and error recovery techniques. In addition, we give model-based SOA a push showing its usefulness to manage self-adaptive systems.

On one hand, in many occasions, the necessary effort to develop and maintain the reliable software intensive systems can be solved by using third-party services. In fact, if we weigh up the cost-effectiveness in terms of the effort required to adapt the system to changes occurred, the best solution is not modifying the code when it is not required, because an intrusive way always requires a reconfiguration of the system that is less efficient, *w.r.t.* time required, than fixing mismatch problems between services by using an adaptor. Regarding this consideration, our proposal always performs with the least effort possible to adapt the system. This is illustrated in our case study, where our approach generated an adaptor in all the situations to fix mismatches and manage context changes. Only in a 50% of cases (driver scenarios A) and B)), our approach needed to modify the system and apply reconfiguration. In a 25% of cases (driver scenario C)), it was necessary to apply error recovery mechanisms.

On the other hand, the development and maintenance of self-adaptive systems using a model-based SOA approach turns out cost-effective. First, because our self-adaptive system provides dependable services to the user, reduces the strong dependence on human resources, and reacts to different events more quickly, being capable of changing its behavior at run-time depending on the context information. Second, due to the model-based SOA facilities, such as integration, interoperability, flexibility, and incorporating of new requirements. Therefore, our model transformation process provides a level of abstraction to tackle discovery, planning, monitoring, adaptation and evolution issues easily and independently of the development platform. Our case study consisted of two kinds of users (with two profiles) and four services in total, so it was not difficult to manage. But, when an organization has a large number of services connected, the management of the service network can become extremely difficult, since all the services are directly connected, which can be unmanageable. In those cases, a model-based SOA may be even more beneficial. A first evaluation to check the scalability of our approach was obtained validating it in several examples with up to 10 services (a booking on-line system, a travel agency, an on-line computer material store, or the case study presented in this work) applied to the dependable composition of services implemented using indistinctly BPEL and WF. In a not far future, we hope that a wide number of companies adopt model-based SOA to definitively bridge the gap between business and information technology, by making the development and maintenance of large software projects more agile.

As regards future work, we plan to develop a full-scale system to check our approach that we successfully applied to a small-scale system. We also want to extend our proposal to deal with security properties in the vector dependencies, by improving the exception management in our fault tolerance mechanism, and tackling in further depth the quality of service. In addition, our approach has some limitations, such as the need of studying how to manage the complexity of the hand-code in case designers must modify the system.

Acknowledgement

This work is partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Science and Innovation (MICINN) and FEDER. The authors are grateful to the anonymous referees who helped to improve the contents and quality of this article.

References

[1] T. Andrews *et al.*. *Business Process Execution Language for Web Services (WSBPEL)*. 2005.

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

[3] H. R. M. Nezhad *et al.* Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*, ACM, 2007.

[4] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, vol. 3324 of *LNCS*, 2004.

[5] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, 2006.

[6] J. Buckley *et al.* Towards a Taxonomy of Software Change. *Software Maintenance and Evolution: Research and Practice*, 17, 2005.

[7] J. Cámara, J.A. Martín, G. Salaün, J. Cubo, 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*, IEEE CS, 2009.

[8] C. Canal, P. Poizat, and G. Salaün. Model-Based Adaptation of Behavioural Mismatching Components. *IEEE Transactions on Software Engineering*, 34, 2008.

[9] J. Cubo, C. Canal, and E. Pimentel. Context-Aware Service Discovery and Adaptation Based on Semantic Matchmaking. In *Proc. of ICIW'10*. IEEE CS, 2010.

[10] J. Cubo *et al.* A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, vol. 215 of *ENTCS*, 2007.

[11] B. Dagenais and M.P. Robillard. Recommending Adaptive Changes for Framework Evolution. In *Proc. of ICSE'08*, ACM, 2008.

[12] A. Dey and G. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *Proc. of Workshop on the What, Who, Where, When and How of Context-Awareness*, 2000.

[13] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[14] 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*, ACM, 2006.

[15] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, ACM, 2004.

[16] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theor. Comput. Sci.*, 138, 1995.

[17] S. Kouadri and B. Hirsbrunner. Towards a Context-Based Service Composition Framework. In *Proc. of ICWS'03*, 2003.

[18] S. KumarGupta *et al.* Backward Error Recovery Protocols in Distributed Mobile Systems: A Survey. *Journal of Theor. and Applied Inform. Technology*, 4, 2008.

[19] T. Mens and S. Demeyer. *Software Evolution*. Springer-Verlag. 2008.

[20] R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. *Theor. Comput. Sci.*, 114, 1993.

[21] O. Nierstrasz *et al.* Model-Centric, Context-Aware Software Adaptation. In *SEAMS*, vol. 5525 of *LNCS*, 2009.

[22] P. Oreizy *et al.* An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14, 1999.

[23] A. Gorbenko *et al.* Experimenting with Exception Propagation Mechanisms in Service-Oriented Architecture. In *Proc. of WEH'08*, ACM, 2008.

[24] F. Tartanoglu *et al.* Dependability in the Web Services Architecture. In *ADS*, vol. 2677 of *LNCS*, 2003.

[25] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Adisson-Wesley, 2nd edition, 2003.

[26] M. Zapletal. Deriving Business Service Interfaces in Windows Workflow from UMM Transactions. In *Proc. of ICSOC'08*, vol. 5364 of *LNCS*, 2008.