Journal of Universal Computer Science, vol. 17, no. 5 (2011), 777-806
submitted: 8/2/10, accepted: 4/11/10, appeared: 1/3/11 © J.UCS

Context-Aware Composition and Adaptation based on Model Transformation

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

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

Ernesto Pimentel
(Department of Computer Science, University of Málaga, Málaga, Spain
ernesto@lcc.uma.es)

Abstract: Using pre-existing software components (COTS) to develop software systems requires the composition and adaptation of the component interfaces to solve mismatch problems. These mismatches may appear at different interoperability levels (signature, behavioural, quality of service and semantic). In this article, we define an approach which supports composition and adaptation of software components based on model transformation by taking into account the four levels. Signature and behavioural levels are addressed by means of transition systems. Context-awareness and semantic-based techniques are used to tackle quality of service and semantic, respectively, but also both consider the signature level. We have implemented and validated our proposal for the design and application of realistic and complex systems. Here, we illustrate the need to support the variability of the adaptation process in a context-aware pervasive system through a real-world case study, where software components are implemented using Windows Workflow Foundation (WF). We apply our model transformation process to extract transition systems (CA-STS specifications) from WF components. These CA-STSs are used to tackle the composition and adaptation. Then, we generate a CA-STS adaptor specification, which is transformed into its corresponding WF adaptor component with the purpose of interacting with all the WF components of the system, thereby avoiding mismatch problems.

Key Words: Reusability, Composition, Adaptation, Model Transformation, Context-Aware Systems, Components, Windows Workflow, Interfaces, Transition Systems.
Category: D.2, D.2.1, D.2.2, D.2.10, D.2.11, D.2.12, D.2.13

1 Introduction

Reuse of software entities (software components, Web services, agents, etc.)[1] is one of the internal factors that determines the quality of software. Thus, component-based and service-oriented systems are developed from the selection, composition and adaptation of pre-existing software entities rather than as a result of programming applications from scratch. Component-Based Software Engineering (CBSE) [Szyperski 2003] and Service-Oriented Architecture

[1] In the sequel, we use the terms component and service indistinctly.

(SOA) [Erl 2005] promote the use of *Commercial-Off-The-Shelf* (COTS) components and services, respectively. Due to their black-box nature, to promote and facilitate their reuse, as well as to enable their automatic composition, components and services must be equipped with rich interfaces enabling external access to their functionality. However, a certain degree of adaptation is required to avoid mismatches, since the interfaces of the constituent components of a system may not be compatible. Several interoperability levels can be distinguished in interface description languages [Canal et al. 2006a]: (i) the *signature level* describes operation names, types of arguments and return values, (ii) the *behavioural or protocol level* specifies the order in which messages for operation invocation are exchanged with their environment, (iii) the *quality of service level* groups other sources of mismatch, usually related to non-functional properties, such as temporal requirements, resources, security, etc., and (iv) the *semantic or conceptual level* is concerned with service functional specifications.

The need to automate these adaptation tasks has driven the development of Software Adaptation (SA) [Yellin and Strom 1997, Canal et al. 2006a]. This discipline manages the interaction between entities by means of *adaptors*, looking for the automation of the adaptation process and enabling components with mismatching to interoperate. The adaptors are automatically built from an abstract description (*adaptation contract*) of how the mismatches can be solved *w.r.t.* the component interfaces. SA is characterised by highly dynamic procedures that occur as devices and applications move from network to network, modifying their behaviour, and enhancing flexibility and maintainability of systems.

Our proposal focuses on composing context-aware mobile and pervasive systems, where devices and applications dynamically find and use components from their environment, providing a semantic representation instead of only a syntactic one. Context-aware computing covers all the topics related to the building of systems which are sensitive to their context (location, identity, time and activity). These systems are different from traditional distributed computing approaches, since some component features may change at run-time depending on the conditions of the environment. Therefore, software adaptation needs to address the issue of quality of service requirements to solve mismatches at runtime, self-adapting those systems to the changing conditions of the environment and the user preferences, thus reducing human effort in the human-computer interaction [Schilit et al. 1994]. On the other hand, there exist different languages focus on Semantic Web technologies, such as Web Services Modeling Ontology (WSMO)[2], METEOR-S [Patil et al. 2004], Semantic Annotations for WSDL and XML Schema (SAWSDL)[3], or Web Ontology Language for Services (OWL-

[2] http://www.wsmo.org/ Accessed on 20 September 2010.
[3] http://www.w3.org/TR/sawsdl/ Accessed on 20 September 2010.

S, formerly DAML-S)[4]. W3C recommends the use of OWL-S to capture the semantic description of components and services. OWL-S is built on the Ontology Web Language (OWL)[5], which proposes a formal representation, expressed in a machine-readable format (XML file), of a set of concepts within a domain by capturing the relationships between those concepts. This is called an *ontology*, and OWL is currently the *de facto* standard for constructing ontologies. For this reason, we use OWL-S ontologies to capture and manage the semantic information of the components. Nevertheless, we could consider other techniques such as Natural Language Processing (NLP)[6] or heuristic-based methodologies [Burton-Jones et al. 2003], preserving the relevance of our proposal.

Considering the aforementioned paradigms, we deal with the reusing of software components based on model transformation. Our contributions (more detailed in Section 2.1) are the following: (i) we propose a model transformation approach that extracts transition systems from component interfaces implemented on existing platforms, and viceversa, (ii) we use verification techniques to validate components against a set of properties, and (iii) we automatically generate an adaptor component addressing all the interoperability levels.

The remainder of this article is organised as follows. Section 2 overviews our proposal and main contributions, as well as a motivating example which will be the case study used throughout the article for illustration purposes. Section 3 presents our context-aware component model, which transforms WF components in CA-STS specifications. Sections 4 and 5 describe the discovery process based on semantic matchmaking and protocol compatibility, and the verification techniques we apply to the CA-STS specifications, respectively. In Section 6, our composition and adaptation process based on model transformation to generate a WF adaptor component from a CA-STS adaptor is presented. Section 7 details some experimental results of applying the developed prototype that implements our approach. In Section 8, we compare our proposal to related work. Finally, in Section 9 some conclusions are drawn and plans for future work are outlined.

2 Overview of the Proposal

In this section, we present our approach and a case study to motivate our proposal, which will be used throughout the article for illustration purposes.

2.1 Approach

Mismatches may occur at the four interoperability levels. However, industrial platforms only deal appropriately with the signature level (e.g. CORBA's IDL[7]).

[4] http://www.daml.org/services/owl-s/ Accessed on 20 September 2010.
[5] http://www.w3.org/TR/owl-ref/ Accessed on 20 September 2010.
[6] http://www.w3.org/TR/nl-spec/ Accessed on 20 September 2010.
[7] http://www.corba.org/ Accessed on 20 September 2010.

Several approaches tackling composition and adaptation based on models focus on the different levels, such as we will compare in Section 8, and aim at generating adaptors which are used to solve mismatch and inconsistences in a non-intrusive way. However, to the best of our knowledge, those approaches do not combine their efforts to tackle the four interoperability levels together. In this article, we focus on all four levels, extending interfaces with context information and a description of their protocol which maintains the conditions. We use ontologies to determine the relationship among the different concepts that belong to a domain. In our previous work [Cubo et al. 2007a], we advocated flexible interaction between an arbitrary number of components depending on the current state of the execution of the system (*i.e.*, current context). These states were defined at design-time. Here, we tackle the need to support the variability of the adaptation process in context-aware systems by continuously reevaluating the context information at run-time. This new adaptation process is based on semantic matchmaking and protocol compatibility mechanisms.

Furthermore, most of those aforementioned approaches abstract from the implementation framework, and few of them relate to existing programming languages and platforms. To the best of our knowledge, the only attempts in this direction have been carried out using CORBA [Gaspari and Zavattaro 1999], COM/DCOM [Inverardi and Tivoli 2003], BPEL [Brogi and Popescu 2006], [Marconi et al. 2009], and SCA components [Motahari et al. 2007]. To relate our model transformation process with realistic and complex examples, we use Windows Workflow Foundation (WF) [Bukovics 2008] developed by Microsoft®. It belongs to the .NET Framework 3.5 and is supported by Visual Studio 2008, interacting with Windows Communication Foundation (WCF) to define component and service interfaces. We have chosen WF because this platform supports behavioural descriptions of components and services using workflows (business processes). In addition, the .NET Framework is widely used in many companies, and WF is increasingly prevalent in the software engineering community [Zapletal 2008, Zapletal et al. 2009]. It makes the implementation of components easier thanks to its workflow-based graphical support and the automation of the code generation. Furthermore, a user could request components or services implemented using WF through mobile devices with Windows Mobile® as the operating system, so our component model could be used in these situations.

Our proposal is based on model transformation, which according to the Model-Driven Architecture (MDA)[8], takes a source model (in our case WF) and produces a target model (in our case transition systems), and viceversa. We illustrate our proposal using a real-world pervasive system as case study. Figure 1 outlines our context-aware composition and adaptation approach, which focuses on systems composed of a component repository, different kinds of users

[8] http://www.omg.org/mda/ Accessed on 20 September 2010.

(clients requesting components)[9], and a shared domain ontology. Our process consists of a set of steps which have been implemented as a set of tools constituting a framework called DAMASCo (Discovery, Adaptation and Monitoring of Context-Aware Services and Components), which is integrated in our toolbox ITACA [Cámara et al. 2009]. When a user performs a request from either a mobile device, a personal computer or a laptop, the composition and adaptation process is executed. First, (1) abstract interface specifications (Context-Aware Symbolic Transition Systems, CA-STSs [Cubo et al. 2009a]) are extracted from the WF components, which implement the client and the components, by means of our model transformation process, and (2) a discovery process based on semantic and compatibility mechanisms finds the WF components satisfying that request. Next, (3) verification techniques are useful in two cases: first, (a) they may help to identify mismatch situations that will determine whether the components involved need adaptation or not, and secondly, (b) they allow validation of a set of properties for the CA-STS client and components selected in (2), by applying symbolic model checking (specifically we use Ordered Binary Decision Diagrams, OBDD [Bryant 1986] because of the use of context information and conditions over transitions). If adaptation is required, then (4) from the discovery process, an adaptation contract is automatically obtained, and (5) being given the CA-STSs corresponding to client and components, as well as the adaptation contract, our process generates a CA-STS adaptor specification, whose resulting composition preserves the properties previously validated in step (3.b). Finally, (6) the corresponding WF adaptor component is generated using our model transformation process, and (7) the whole system is deployed, allowing the WF client and components to interact via the WF adaptor. This article tackles the reusing of software components and makes the following contributions:

– We propose a model transformation approach that extracts transition systems from component interfaces implemented on existing programming languages/platforms, and viceversa. Our model extends [Mateescu et al. 2008] and improves substantially [Cubo et al. 2009a, Cubo et al. 2009b] by considering context information, and supporting conditions to control the execution of the protocols according to certain changes at run-time. Users can execute several requests simultaneously (concurrent interactions), and the components can be instantiated several times.
– Verification techniques are used to validate if the client and the components are free of mismatches and inconsistences (with respect to a set of properties). We have redesigned the algorithms presented in [Cubo et al. 2009b] with respect to this extended version of our model.
– We automatically obtain the adaptation contract by using similarity of context information, and mechanisms based on both semantic matchmaking

[9] We distinguish between clients and components with the purpose of the comprehension -clients requesting components- although both refer to components or services.

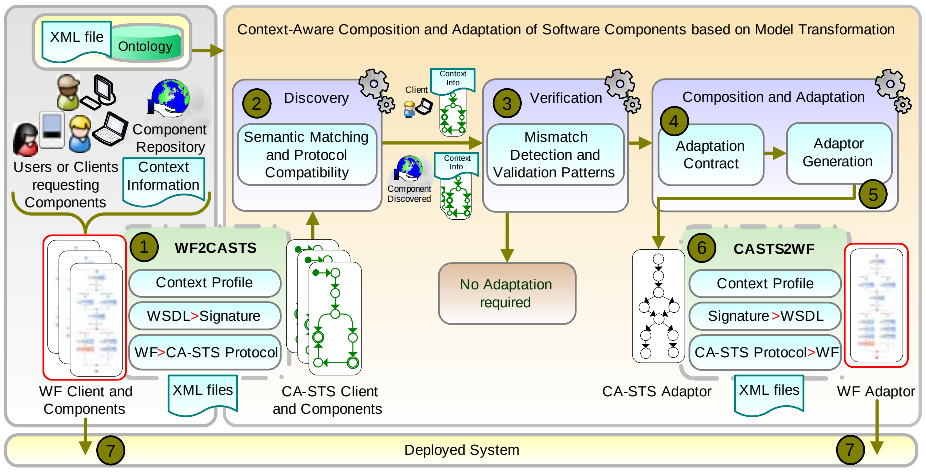

Figure 1: Overview of our Context-Aware Composition and Adaptation approach

of the operations and protocol compatibility. The adaptor component generated considers not only the signature and behaviour levels of the components, but also the quality of service and semantic levels. Therefore, our proposal generates an adaptor addressing all the interoperability levels.

2.2 Motivating Example: On-line Booking System

To illustrate our proposal, we describe an on-line booking system consisting of clients and a component repository. Clients can perform different requests; they can book a restaurant and/or a taxi by means of either a mobile device, a personal computer or a laptop. We assume the component repository contains components named Restaurant, Restaurant Database (or simply RestDB) and Taxi. When a user executes several requests simultaneously or a reasonable number of users make requests concurrently, instances of the requested components are generated for each request. The system considers the context information given by the client, which is not sent explicitly, since components infer it from the client request (through the HTTP header of SOAP messages [Cubo et al. 2009b]). Thus, the Restaurant component can receive a client request to find restaurants close to a particular address, taking into account the context information related to the client privileges. We assume the system accepts two client profiles which defines the client privileges: (i) *"VIP"* refers to clients with a certain priority (memberships paying a fee to access to the system), or (ii) *"Guest"*, *i.e.*, regular clients. Depending on the privileges of a particular client, the list of restaurants returned will be different. Once the client knows the restaurants in the vicinity of that address, he/she can book one of them. The Restaurant component uses the Restaurant Database (RestDB) component to check whether the selected restaurant has a table available for a given date and a number of persons. After these interactions, the Restaurant component may receive a booking message and send an acknowledgement back. On the other hand, the Taxi component receives a client request to book a taxi, with a destination address and the context information related to the client location and his/her privileges (both inferred from the client request). The client will book a taxi only if the price does not exceed a limit amount, and he/she can either pay on reservation or later on. In the case where the client pays the taxi immediately, the Taxi component will send a receipt in the correct format depending on the context information corresponding to the client device (mobile or PC/laptop), and the client language. This case study corresponds to a context-aware pervasive system in which the client location and profile can change at run-time. Depending on such variations, the system must adapt to work correctly in any situation. For the sake of comprehension, in the remainder of this article we focus on a single session which corresponds to the connection and use of the system by one client. An example handling any number of sessions can be derived from our simplified version, generating instances of the requested components.

In next section, we present the implementation of this example using the WF platform. Then, we describe our formal model to manage the variability and transform WF components into CA-STS specifications.

3 Context-Aware Component Model

Several platforms or languages already exist and can be used to implement components and services, such as UML[10], BPEL[11] or WF. To illustrate our model transformation approach we have chosen WF because it is an interesting alternative compared to the others available. Nevertheless, we have also validated our proposal using BPEL. In this section, we describe a formal model for the WF component interfaces 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. Furthermore, any formalism to describe dynamic behaviour may be expressed in terms of a transition system [Foster et al. 2006]. Thus, our approach becomes general enough to be applied in other contexts.

3.1 Abstraction of WF Workflows

In order to illustrate our example, we use a representative kernel of the WF activities, namely `Code`, `Sequence`, `Terminate`, `Receive`, `Send`, `IfElse`, `While`, and `Listen` with `EventDriven` activities. These activities are enough to understand the example. In Table 1, we formalise the grammar defined for a textual

[10] http://www.uml.org/ Accessed on 20 September 2010.
[11] http://www.oasis-open.org/committees/wsbpel/ Accessed on 20 September 2010.

notation (left side) of the WF activities considered (on the right the meaning of these activities is provided), which abstracts several implementation details. Our grammar considers as input textual workflows (defined in XML) corresponding to the graphical description of the WF workflows, with WF activities $\mathcal{A}$, where C, C_i are boolean conditions, I, I_i (inputs), O, O_i (outputs) are parameters of activities, and Id are component or service identifiers.

$\mathcal{A}$ ::=	`Code`	*executes a chunk of code*
|	`Terminate`	*ends a workflow's execution*
|	`Receive`$(Id,Op[,O,I_1,\ldots,I_n])$	*receives a msg from a component/service*
|	`Send`$(Id,Op[,O_1,\ldots,O_n,I])$	*sends a msg from a component/service*
|	`Sequence`$(\mathcal{A}_1,\mathcal{A}_2)$	*executes first $\mathcal{A}_1$ and then $\mathcal{A}_2$*
|	`IfElse`$((C_1,\mathcal{A}_1),\ldots,(C_n,\mathcal{A}_n),\mathcal{A}_{n+1})$	*executes $\mathcal{A}_i$ if C_i is true, $\mathcal{A}_{n+1}$ otherwise*
|	`While`$(C,\mathcal{A})$	*executes $\mathcal{A}$ while C is true*
|	`Listen`$(E_1,\ldots,E_n)$	*fires one of the E_i branches*
E ::=	`EventDriven`(`Receive`$(Id,Op[,I_i])$,$\mathcal{A}$)	*executes $\mathcal{A}$ when Id is received*

Table 1: Grammar for the abstract notation of WF workflow activities

WF platform has capabilities for developing workflows in different scenarios, from simple sequential ones to realistic and complex state machine-based workflows with human interaction. The programming languages available are *Visual Basic* and *C#*. Our example have been implemented in *C#*.

Example. We have designed WF workflows for the Client-restaurant and Client-taxi requests, and for the Restaurant, RestDB and Taxi components. WF provides a WSDL description for each WF workflow. In Figure 2, we show the WF workflow that represents the behaviour of the Client-restaurant request. For space reasons, we do not depict the others WF workflows of our example. In the workflow, the message names prefixed with `send` and `receive`, such as

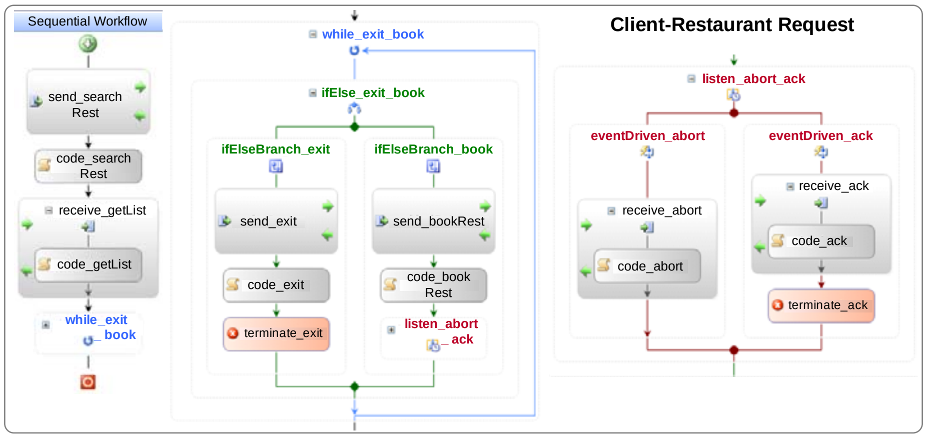

Figure 2: WF workflow corresponding to the Client-restaurant request

`send_searchRest` and `receive_getList`, represent a `Send` and a `Receive` activity, respectively. Those prefixed with `code`, such as `code_getList`, correspond to the execution of *C#* code. The `while_exit_book` label denotes that the client will enter in a loop where the `ifElse_exit_book` label indicates that the client will exit if the branch condition `ifElseBranch_exit` is true (*[list==null]*), or he/shell will begin the booking of a restaurant if the branch condition `ifElseBranch_book` is true (*[list≠null]*). Conditions are mutually exclusive, and we have previously defined their ranges of values. In the booking process, the client will receive either an abort, `receive_abort` message, because of the unavailability of tables in the selected restaurant or an acknowledgement of the booking `receice_ack`. This is controlled by means of the listen activity `listen_abort_ack` with two `EventDriven` activities, `eventDriven_abort` and `eventDriven_ack`. The activities `terminate_exit` and `terminate_ack` conclude the client session. To illustrate our textual notation, we focus on the `Listen` construct.

```
Sequence(...,Listen(EventDriven(Receive(receive_abort, abort),Code),
          EventDriven(Receive(receive_ack, ack) ,Sequence(Code,Terminate))))
```

Note in the abstract notation we remove the suffixes used in the workflow (except those related to identifiers) to distinguish activity names.

3.2 CA-STS Interface Model

This section introduces our model to describe component interfaces, which are given using context information, a signature, and a behavioural or protocol description, taken into account through the use of abstractions of WF workflows.

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"* [Dey and Abowd 2000]. We consider context information from both a user-centric and a service-centric point of view, by allowing interaction between user requirements and service capabilities depending on run-time changes of both user and service contexts, and discovering component or services according to certain non-functional properties. Context information can be represented in different ways, by including complex functions involving required properties [Mostéfaoui and Hirsbrunner 2003]. For our purpose, we only need a simple representation where contexts are defined by context attributes with associated values. Thus, we represent the component 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 that CA is public or private (*e.g.*, (*priv, Guest, dynamic, public*)). Both clients and components are characterised by public (*e.g.*, weather, temperature, ...) and private (*e.g.*, personal data, bandwidth, ...) context attributes. We also differentiate between static context attributes (*e.g.*, role, day, ...) and dynamic ones (*e.g.*, network connectivity, location, ...), because the latter can change continuously at run-time, therefore they have to be dynamically evaluated during the composition. On the other hand, 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 Labelled Transition System (LTS) 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 to determine the relationship among the different concepts that belong to a 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 common variables or static context attributes, and variables 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 a tuple (B, M, D, F) representing an event, where: B is a condition (represented by a boolean expression), 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 corresponding 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, and $T \subseteq S \times A \times S$ is the transition function whose elements $(s_1, a, s_2) \in T$ are denoted by $s_1 \xrightarrow{a} s_2$.

Finally, a *CA-STS interface* (or *CA-STS specification*) is constituted by a tuple (CP, SI, P), where: CP is a context profile, and SI is the signature corresponding to a CA-STS protocol P.

Both client and components consist of a set of interfaces, since we assume they have several protocols with their corresponding signatures, and a context profile for each one. For instance, in our example a client may perform two different requests. The client consists of two interfaces: one for the Client-restaurant request $CR = (CP_{CR}, SI_{CR}, P_{CR})$ and the other one for the Client-taxi request $CT = (CP_{CT}, SI_{CT}, P_{CT})$, as is depicted in Figure 5. In the Client-restaurant request, CP_{CR} refers to the context information *priv*, SI_{CR} is composed by all the operation profiles, such as $l_{cr_1} = searchRest!address, \widetilde{priv}$, and P_{CR} is the protocol which indicates the behaviour of the CA-STS. For example, l_{cr_1} means that a client with the context information related to privileges *priv* issues an emission looking for a restaurant in a specific *address*, and then this client receives a list of restaurants found $l_{cr_2} = getList?list$, and so on. Initial and final states are respectively depicted in CA-STSs using bullet arrows and hollow states. Our proposal is suitable for synchronous systems where clients interact with composite components (client/server model), such as mobile systems. Therefore, we adopt a synchronous and binary communication model, where clients can execute several protocols simultaneously (concurrent interactions), and client and component protocols can be instantiated several times. Note that we consider binary communication because of the composite components.

At the user level, client and component 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 or abstraction of WF workflows, for protocols. In this work, we assume context information is inferred by means of the client requests (header of SOAP messages), and we consider processes (client and components) implemented as WF workflows which provide the WSDLs and the protocol descriptions.

3.3 Extracting CA-STSs from WF Components

CA-STSs are used as an abstraction to focus on behavioural composition issues by describing component interfaces in a standard notation. These CA-STSs are automatically generated from the WF components. For each WF component, our model transformation process parses the three XML files corresponding to its context information, WSDL description, and WF workflow. A new XML file containing 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 in a prototype tool, called `WF2CASTS`, following our model transformation process presented in Figure 3.

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 viceversa. 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 is not 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:

- `Code` is internal and hence interpreted as an internal transition;
- `Terminate` corresponds to a final state;
- `Receive` and `Send` are reception and emission, respectively;
- `Sequence` is translated to preserve the order of the involved activities. Final states of the first activity are linked to the initial state of the second one;

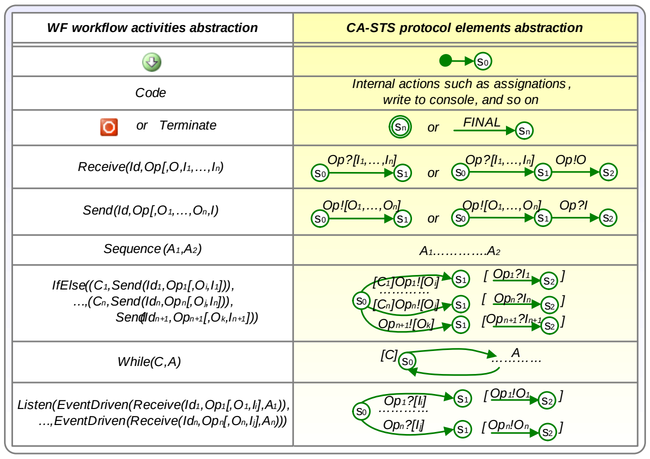
WF workflow activities abstraction
CA-STS protocol elements abstraction
S0
Code
Internal actions such as assignations, write to console, and so on
or Terminate
Sn or FINAL Sn
Receive(Id,Op[,O,I1,…,In)
Op?[I1,…,In] or Op?[I1,…,In] Op!O
Send(Id,Op[,O1,…,On,I)
Op![O1,…,On] or Op![O1,…,On] Op?I
Sequence (A1,A2)
A1…………A2
IfElse((C1,Send(Id1,Op1[,Oi,I1])), …,(Cn,Send(Idn,Opn[,Oj,In])), Send(Idn+1,Opn+1[,Ok,In+1]))
[C1]Op1![Oi] [Op1?I1]
[Cn]Opn![Oj] [Opn?In]
Opn+1![Ok] [Opn+1?In+1]
While(C,A)
[C] A
Listen(EventDriven(Receive(Id1,Op1[,O1,Ii],A1)), …,EventDriven(Receive(Idn,Opn[,On,Ij],An)))
Op1?[Ii] [Op1!O1]
Opn?[Ij] [Opn!On]
S0 S1 S2

Figure 3: Patterns of our Model Transformation process from WF workflow activities abstraction to CA-STS protocol elements abstraction and viceversa

– `IfElse` corresponds to an internal choice. This corresponds to as many transitions as there are branches in the `IfElse` construct (even the else). Each of these transitions leads to the initial state of the corresponding activity;
– `While` is translated as a looping behaviour, where the condition determines the choice between termination or loop;
– `Listen` corresponds to an external choice. This corresponds to as many outgoing transitions as there are branches in the `Listen` construct. These transitions are labelled with receptions corresponding to the messages that can be received and target the initial state of the related activity.

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 components of our scenario in order to obtain the corresponding CA-STS specifications. To facilitate comprehension, Figure 4 illustrates the transformation process which generates CA-STS internal (emissions) and external (receptions) choices (with the corresponding states, transitions and labels), from WF `IfElse` and `Listen` activities, respectively. Figure 5 shows the whole interfaces corresponding to the Client-restaurant and Client-taxi requests (CR and CT, respectively), and the Restaurant (R), RestDB (RD) and Taxi (T) components. Each interface has a context profile, a signature and a CA-STS protocol. With the purpose of the comprehension, we present the signature corresponding to the Client-restaurant request: *searchRest!(string,Tpriv); getList?(Tlist); bookRest!(string,DateTime,int); ack?(); abort?(); exit!()*.

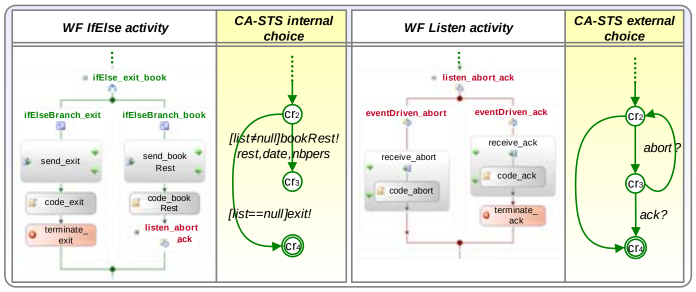

Figure 4: Transformation Process of WF `IfElse` and `Listen` activities

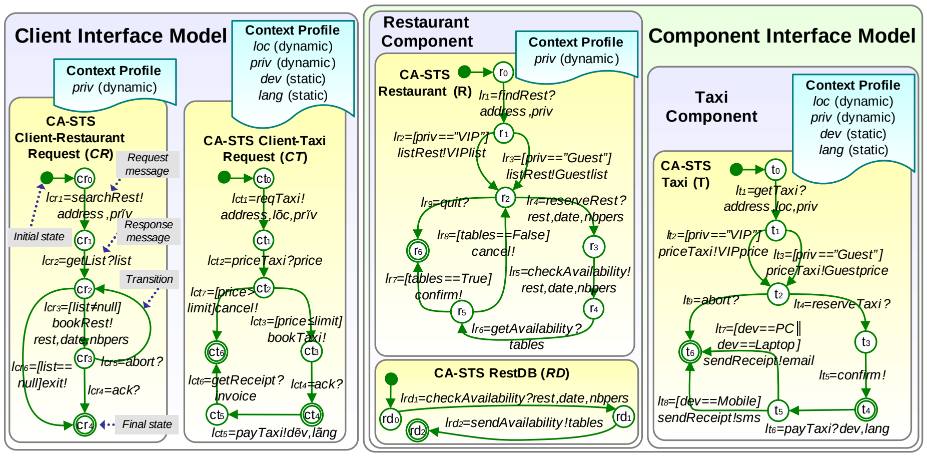
Client Interface Model
Context Profile
loc (dynamic)
priv (dynamic)
dev (static)
lang (static)
CA-STS Client-Restaurant Request (CR)
CA-STS Client-Taxi Request (CT)
Request message
Response message
Initial state
Transition
Final state
lcr1=searchRest! address,priv
lcr2=getList?list
lcr3=[list≠null] bookRest! rest,date,nbpers
lcr4=ack?
lcr5=abort?
lcr6=[list==null]exit!
lct1=reqTaxi! address,loc,priv
lct2=priceTaxi?price
lct3=[price≤limit] bookTaxi!
lct4=ack?
lct5=payTaxi!dev,lang
lct6=getReceipt? invoice
lct7=[price>limit]cancel!
Restaurant Component
CA-STS Restaurant (R)
lr1=findRest? address,priv
lr2=[priv=="VIP"] listRest!VIPlist
lr3=[priv=="Guest"] listRest!Guestlist
lr4=reserveRest? rest,date,nbpers
lr5=checkAvailability! rest,date,nbpers
lr6=getAvailability? tables
lr7=[tables==True] confirm!
lr8=[tables==False] cancel!
lr9=quit?
CA-STS RestDB (RD)
lrd1=checkAvailability?rest,date,nbpers
lrd2=sendAvailability!tables
Component Interface Model
Taxi Component
CA-STS Taxi (T)
lt1=getTaxi? address,loc,priv
lt2=[priv=="VIP"] priceTaxi!VIPprice
lt3=[priv=="Guest"] priceTaxi!Guestprice
lt4=reserveTaxi?
lt5=confirm!
lt6=payTaxi?dev,lang
lt7=[dev==PC‖ dev==Laptop] sendReceipt!email
lt8=[dev==Mobile] sendReceipt!sms
lt9=abort?

Figure 5: CA-STS specifications of Client, Restaurant, RestDB and Taxi

Our CA-STS protocols maintain conditions appearing in WF `While` and `ifElse` constructs, since our interface model considers them. Thus, transitions tagged with conditions must assure their mutual exclusion and that they control the full range of possible values. For instance, if we consider the conditions *[list==null]* and *[list≠null]* of the CA-STS Client-restaurant request *CR*, because of *Tlist* definition, it is trivial to see that both conditions are mutually exclusive. In the case of the conditions *[priv==“VIP”]* and *[priv==“Guest”]* of the CA-STS Restaurant *R*, the range of values of the type *Tpriv* ({*“VIP”*, *“Guest”*}) has been defined previously, and we can determine that both are mutually exclusive.

4 Semantic-Based Component Discovery

We propose a context-aware component discovery process, that consists of two steps. First, we perform a semantic matchmaking process that selects the most appropriate component interfaces for a client request depending on their contexts and signatures. Second, an analysis of the behavioural part of the model determines whether two protocols are compatible. Our process establishes a ranked list of the component interfaces that better match the client request.

Semantic Matchmaking. This process is based on OWL-S ontologies. Thus, context attributes included in a context profile and operation profiles of a signature refer to OWL-S concepts with their associated semantics. Our goal is to measure the semantic matchmaking of context attributes and operation profiles. The semantics of OWL-S descriptions allows to define a ranking function which distinguishes multiple degrees of match between two OWL-S concepts. We choose the notion of degree of match introduced in [Paolucci et al. 2002] to define the semantic matchmaking based on ontologies.

Definition 3 Degree of Match. Being c_r and c_p either two context attributes, operation names, arguments, or return types, that respectively belong to a requester (client or component interacting with another component) and a provider interface, we define their degree of match with in an ontology *Ont* as follows:

$$degree\_match(c_r, c_p, Ont) = \begin{cases} \texttt{exact} & \text{if } (c_r = c_p \vee c_r \textit{ subclass of } c_p) \text{ in Ont} \\ \texttt{plugIn} & \text{if } (c_p \textit{ subsumes } c_r) \text{ in Ont} \\ \texttt{subsume} & \text{if } (c_r \textit{ subsumes } c_p) \text{ in Ont} \\ \texttt{fail} & \text{otherwise} \end{cases}$$

The order of matching patterns is the following: 1) `exact`, 2) `plugIn` because the context attribute, operation name, argument or type of the provided component can be used in place of the one that is expected by the requester, 3) `subsume` indicates that the requirements of the requester are only partially satisfied, and 4) `fail` represents an unacceptable result. This order is determined by the function $\sqsupset$ that compares two degrees of match: `exact` $\sqsupset$ `plugIn` $\sqsupset$ `subsume` $\sqsupset$ `fail`. We discretise each degree of match to a numeric value (`exact`=3, `plugIn`=2, `subsume`=1, `fail`=0) to obtain the average among several degrees of match. Then, this average value is rounded (upward if the first decimal number is greater than 5 or downward in otherwise) to be transformed into its respective degree of match that determines the final degree of the average (*e.g.*, the average value of `exact` and `subsume` is 2, whose degree of match is `plugIn`).

This process have been implemented in a prototype tool, `ConTexTive`, which takes as input a requester (client or component) interface, a list of provider component interfaces corresponding to the available components into a repository, and a shared domain ontology. First, the process selects the component interfaces whose context attributes are similar to the requester ones. Then, considering these pre-selected component interfaces, the process returns a list with the candidate component interfaces that satisfy the request at the signature and semantic levels, and their corresponding matching tuple set. The list contains the respective composite component interfaces. Each matching tuple set consists of tuples which constitute an operation profile from the requester interface, the operation profile from the provider component interface which best matches the requester one and the degree of match for both operation profiles.

Example. Figure 6 gives the shared domain ontology related to the on-line booking system. We present the classes used in our scenario with their relationships. These classes represent concepts which may be either a context attribute, an operation name, or an argument (return types are not represented to simplify). This ontology has been generated graphically using Protégé 4.0.2[12], obtaining directly the corresponding XML file.

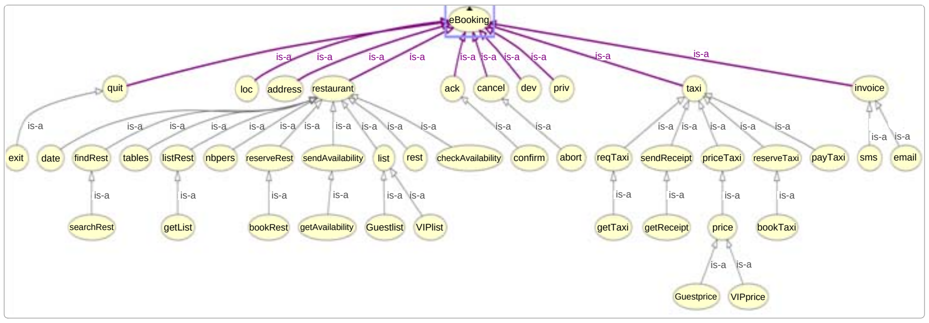

Figure 6: On-line Booking System Ontology generated using Protégé 4.0.2

The semantic matchmaking process is executed individually for each client interface: restaurant and taxi requests. Let us focus on the Client-restaurant request CR. First, the process uses the function $degree\_match$ to select the component interfaces whose context attributes are similar to the client ones, considering also those interfaces whose context profiles are empty. Restaurant R and Taxi T components are selected, since both include the context attribute $priv$ of Client among theirs context attributes. RestDB RD component is also selected, because its context profile is empty and our process needs to know more information about it. Next, our process computes the semantic matchmaking of the operation profiles, and it determines that only the signature of R matches semantically the CR. In the following, we present the matching tuple sets calculated by the process for the interaction of the Client-restaurant interface CR with the Restaurant component interface R:

$$MT_{CR,R} = \{(l_{cr_1}, l_{r_1}, \texttt{exact}), (l_{cr_2}, l_{r_2}, \texttt{exact}), (l_{cr_2}, l_{r_3}, \texttt{exact}), (l_{cr_3}, l_{r_4}, \texttt{exact}), (l_{cr_4}, l_{r_7}, \texttt{subsume}), (l_{cr_5}, l_{r_8}, \texttt{exact}), (l_{cr_6}, l_{r_9}, \texttt{exact}), (\_, l_{r_5}, \_), (\_, l_{r_6}, \_)\}$$

This set indicates that, for instance, in the first matching tuple, the degree of match for the operation profiles l_{cr_1} and l_{r_1} (represented in Figure 5) with respect to the ontology (presented in Figure 6) is `exact`. This is calculated as follows: (i) the degree of match of the operation names $searchRest$ and $findRest$ is `exact`, (ii) the average of the degree of match between theirs arguments and types is `exact`, and (iii) the degree of match for that matching tuple is the average between both degrees of match, $i.e.$`exact`. This process is performed for the combination of l_{cr_1} with all the operation profiles of R whose directions are reversed

[12] http://protege.stanford.edu/ Accessed on 20 September 2010.

(binding of emissions and receptions) to the direction of l_{cr_1}, and the matching process determines that the best one is l_{r_1}. Then, the process is executed for each operation profile in *CR* obtaining the rest of the matching tuples. Labels may have several correspondences in case the degree of match is equal (*e.g.*, l_{cr_2} matches l_{r_2} or l_{r_3} according to the value of the context attribute *priv*). When any operation profile of the requester/provider interface does not have counterpart (open operation profile or open port) in the provider/requester interface, then the process is again executed with the interface(s) with open port(s) as requester(s), but only looking for the operation profiles marked as open. This is represented in the matching tuple sets with the symbol "_". For instance, the operation profiles l_{r_5} and l_{r_6}, which belongs to R, are open, because they do not match any operation profile of *CR*. Therefore, Restaurant R needs to discover the best interfaces connecting those operation profiles which have no counterparts. Now, the process takes as input R as requester interface, the rest of the component interfaces as possible providers (RD and T) and the ontology. Firstly, the process selects Taxi T component, since it includes the context attribute *priv* of Restaurant R among their context attributes, and RestDB component RD, whose context profile is empty. The second phase determines that only RD matches R semantically, with the matching tuple:
$MT_{R,RD} = \{(l_{r_5}, l_{rd_1}, \texttt{exact}), (l_{r_6}, l_{rd_2}, \texttt{exact})\}$

Therefore, the process determines the composite component Restaurant R and RestDB RD match semantically with Client-restaurant request *CR*.

The same procedure is performed to obtain the matching tuple set for the Client-taxi request *CT*, which is not presented in this article for space reasons.

Protocol Compatibility. Semantic matchmaking techniques are not enough to ensure the component compatibility, since the presence of incompatibilities in component interactions may result in a deadlocking execution of any component.Therefore, `ConTexTive` also checks the compatibility between protocols. There exist different notions of compatibility in synchronous communication, such as opposite behaviours, unspecified reception, and deadlock-freeness [Bordeaux et al. 2004]. We choose 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 protocols are performed in a satisfactory way, leading to a correct final state. To verify this protocol compatibility notion, first, the *synchronous product* of CA-STS protocols is computed, by obtaining a new CA-STS protocol that contains all the possible interactions between the involved components. Although conditions are kept in this new CA-STS protocol, they are not used to build the product because they cannot be evaluated. Then, we check that the product is free of deadlock mismatch.

Definition 4 CA-STS Synchronous Product. The *Synchronous Product* of n CA-STSs $P_i = (A_i, S_i, I_i, Fc_i, T_i)$, $i \in \{1, \ldots, n\}$, *w.r.t.* an ontology *Ont*, is the CA-STS $P_1||\ldots||P_n$ characterised by a tuple (A, S, I, Fc, T) such that:

– $A = A_1 \times \ldots \times A_n$, $S = S_1 \times \ldots \times S_n$, $I = (I_1, \ldots, I_n)$, $Fc = Fc_1 \times \ldots \times Fc_n$, where the $\times$ operator stands for the cartesian product, and
– T is defined as: given $(s_1, \ldots, s_n) \in S$, and $i < j$ such that $(s_i, [b_i]a_i, s'_i) \in T_i$, $\left(s_j, [b_j]a_j, s'_j\right) \in T_j$, where $a_i = m_i!e_i$ and $a_j = m_j?e_j$, $type(e_i) = type(e_j)$, and $operation\_matching(a_i, a_j, Ont)$, then:
$((s_1, \ldots, s_n), [b_i \wedge b_j](a_i, a_j), (s_1, \ldots, s'_i, \ldots, s'_j, \ldots, s_n)) \in T$

Function *operation_matching* computes the degree of match for two operation profiles, considering operation names, arguments and return types. The states in the product correspond to sets of states of the components. The initial state of the product is $(I_1, \ldots, I_n)$, since initially all components are in their initial state. The transitions mean that, given some composite state $(s_1, \ldots, s_n)$ in the product, there is some transition outgoing from this state *iff* there are two components, i and j, that may perform at the same time, from states s_i and s_j, in their respective CA-STSs, complementary events (*i.e.*, one sending a message and the other one receiving it). Next, we formalise the concepts of deadlock mismatch and protocol compatibility for CA-STS.

Definition 5 Deadlock Mismatch. A CA-STS protocol $P = (A, S, I, Fc, T)$ presents a deadlock mismatch, if there is a state $s \in S$ such that $s \notin Fc$ and there are no outgoing transitions from s, *i.e.*$(s, l, s') \notin T$. $\forall l \in A, s' \in S$. Such state s is denoted by $dead(s)$.

Definition 6 Deadlock-freeness Compatibility. Two CA-STS protocols P_1 and P_2 are *deadlock-free compatible* or *protocol compatible*, if their synchronous product $P_1||P_2$ is free of deadlock mismatches.

Example. In our example, we have to check the compatibility of *CR* and the composite component R with *RD*. To do so, our protocol compatibility process generates the synchronous product for the interactions between protocols (Figure 7) and verifies that no deadlock mismatches exist in their synchronous product, thus protocols are deadlock-free compatible.

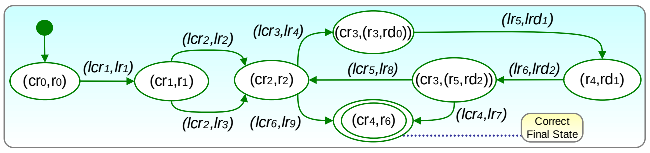

Figure 7: Deadlock-free Synchronous Product of *CR* with R and *RD* Protocols

5 Verification of the Component Model

This section presents verification techniques that check if there exist mismatch situations and that validate a set of properties for each CA-STS protocol by using symbolic model checking, specifically OBDD (Ordered Binary Decision Diagrams). Verification techniques are useful in two cases: first, they may help to identify behavioural mismatch situations that will determine whether the components involved need adaptation or not, and second, they allow to validate a set of properties for the CA-STS client and components discovered in Section 4.

Mismatch Detection. First of all, using the matching tuple sets, we need to check if there exist behavioural mismatches in the interaction among client and components. Thus, if a mismatch exists, we apply our composition and adaptation process that aims at generating a CA-STS adaptor specification (Section 6). **Example.** In our scenario we find mismatches such as different message names (*e.g.*, *searchRest!* in *CR* versus *findRest?* in *R*), or label correspondences depending on the value of the context information (*e.g.*, l_{cr_2} can correspond with l_{r_2} or l_{r_3} depending on the value of the context attribute *priv*). Therefore, the components cannot be directly used together, and an adaptor component has to be incorporated as a third-party in the interaction.

OBDD Verification Model. Before performing the composition and adaptation process, we need to validate each CA-STS to assure they are free of inconsistences. This is required because of the inclusion of context information and conditions in our component interface model. Therefore, our OBDD verification model validates our context-aware component model against a set of properties. The communication between client and components is supposed to be a finite flow of requests/responses in which context information is used to improve the provided component. Next, we describe the validation patterns we consider:

- Determinism: in each state in which the computation can follow different paths, conditions on those multiple requests/responses must be mutually exclusive. If two conditions would be satisfiable simultaneously then the result would be non-deterministic and the result will depend on the implementation and not on the context value itself.
- State liveness: if in a state context values are used to select the next request/response, at least one combination of values of those contexts must lead to a transition. It requires that at least one outbound transition is enabled for each state (with the exception of the final state).
- Request/response liveness: if a request/response is conditioned by a certain value of the context, that condition must be satisfiable. It guarantees that all the specified transitions will be satisfiable for at least one combination of context values. A condition like *[priv==“VIP”* && *priv==“Guest”]* will never be satisfied and the corresponding transition will never be executed.
- Non-blocking states: irrespective of the values of the context variables, communication should always reach a final state. The absence of non-blocking states guarantees that independently from the values of the context, it should be possible to continue the communication avoiding deadlocks.

These properties are verified for the CA-STS protocols related to client and components. Once CA-STS adaptor is generated (Section 6), we could apply our OBDD verification model as well. However, it will not be needed, since the verification model assures that the properties are preserved in the composition of the CA-STSs already validated, *i.e.*, in the CA-STS adaptor. To validate these properties, we have designed an OBDD representation of our CA-STS protocols. We selects OBDD because it has been shown that in many circumstances OBDDs offer a compact way to represent and manipulate Boolean functions. Our idea here is to show how states and labels can be represented by means of conjunctions of Boolean variables, and transition relations can be encoded by mens of Boolean formulae, which are manipulated using OBDDs.

As described in Section 3.2, let (A, S, I, Fc, T) be a CA-STS protocol. The number n of Boolean variables required to encode the set of states S is $n = \lceil log_2|S|\rceil$. Thus, if S contains 7 states, then 3 Boolean variables $\{s_1, s_2, s_3\}$ are required. We represent these variables by means of a Boolean vector $\overline{s} = (s_1, \ldots, s_n)$ where each $s_i \in \overline{s}$ can take either the value 0 (negation of a variable) or 1. For instance, the first state in S could be identified by the vector $(1, 1, 1)$, the second state by the vector $(1, 1, 0)$, and so on. Correspondingly, the first state is encoded by the Boolean formula $s_1 \wedge s_2 \wedge s_3$, the second state by $s_1 \wedge s_2 \wedge \neg s_3$ (notice the last negation), and so on[13]. Sets of states are encoded by Boolean formulae as well, by taking the disjunction of the Boolean formulae encoding each state in the set. Thus, the set of states composed by the first and the second state of S is represented by $(s_1 \wedge s_2 \wedge s_3) \vee (s_1 \wedge s_2 \wedge \neg s_3)$. Consider a message $a = (b, m, d, f) \in A = (B, M, D, F)$ encoded as a Boolean formula by associating a Boolean variable to each expression or variable in f, and representing the condition b by means of these variables. We denote with $\overline{a}$ the Boolean encoding of a given message $a \in A$. Having encoded states and messages, we can encode the transition relation T by introducing a set of "primed" variables $(s'_1, \ldots, s'_n)$ to encode the destination state of a transition. A transition $s \xrightarrow{a} t$ is encoded by means of the Boolean formula $\overline{s} \wedge \overline{a} \wedge \overline{t}$, where the overlined variables denote Boolean expressions and $\overline{t}$ is encoded in terms of the primed variables. The whole transition relation T is encoded as a Boolean formula by taking the disjunction of all the elements of T, *i.e.,* $\overline{T} = \bigvee_{s \xrightarrow{a} t \in T} (\overline{s} \wedge \overline{a} \wedge \overline{t})$.

We have implemented a set of OBDD-based algorithms [Cubo et al. 2009b] to verify the set of properties. `CASTS2OBDD` is a prototype tool that implements those algorithms by representing CA-STS protocols by means of OBDD.

Example. Our verification model determined our client and components are free of inconsistences with respect to the validation patterns considered. This process is not obvious at a first analysis, but the verification mechanisms we apply can

[13] By slight abuse of notation, the same symbols s_i are used to denote Boolean variables or their value in a vector, and atomic propositions in logical formulae.

check automatically the set of properties previously defined. For instance, if we consider the CA-STS protocols corresponding to the Client-restaurant request CR, Restaurant R and RestDB RD, the verification of those properties required less than 1 second for all those CA-STSs. We employed up to 7 Boolean BDD variables (*i.e.*, 3 for CR, 3 for R, and 1 for RD) to encode our scenario, corresponding to a model of size 2^7. Notice that the properties presented above could not be checked using a standard model checker, because of the use of conditions over transitions and because our requirements reason about these conditions over transitions. Indeed, a standard model checkers only allows to reason about (sequences of) states by means of temporal formulae.

6 Context-Aware Composition and Adaptor Generation

In this section, we present our context-aware composition and adaptation process, which is performed once we have checked the CA-STS client and components of the system are free of inconsistences. Our process defines correspondences between client and component messages considering context information. From the client, a set of components and an adaptation contract, a CA-STS adaptor can be automatically generated. Using our transformation process we obtain a WF adaptor from the CA-STS one, which will work as an intermediate piece connecting the client and components to interact in a particular case.

6.1 Adaptation Contract

An adaptation contract is built as a set of correspondences among messages of the involved components. From the matching tuple sets obtained in our discovery process, we can automatically generate an adaptation contract when there exist mismatches in the interaction between client and components. Moreover, apart from solving some cases of behavioural mismatches, we want composition and adaptation to distinguish between the available contexts when translating the messages among components. Using a non-contextual approach, message correspondences are fixed, which means that any client request is always associated to the same target message. This prevents changes in these connections being taken into account, and motivates the need for new capabilities that our context-aware composition and adaptation approach provides in order to achieve message translation depending on contexts. We have to define that contract between events in the CA-STS protocols. Here we define a notation based on vectors expressing correspondences among component messages, and on transition systems to specify the evolution of every component depending on its contexts. These interactions are formalised through *synchronisation vectors* [Arnold 1994] which allow messages with different names and even different numbers of messages to be synchronised. Thus, a vector does not require interactions on the same names of events as it is the case in process algebra for instance. The interactions denote a communication among several components. Each event (or label) appearing in one vector is executed by one component, and the overall result corresponds to a synchronisation between all the involved components.

Definition 7 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 $\langle l_1, \ldots, l_n \rangle$ with $l_i \in A_i \cup \{\varepsilon\}$, ε meaning that a component does not participate in a synchronisation.

In some situations it is essential to apply a specific order between vectors to avoid mismatches. Hence, we use as abstract notation for our composition an LTS with vectors on transitions. This LTS is used as a guide in the application order of interactions denoted by those vectors. Our approach gives the user assistance to generate this LTS. We model the component composition by introducing the notion of adaptation contract, that makes use of vectors and vector LTS .

Definition 8 Adaptation Contract. An adaptation contract for a set of components C_i, $i \in 1, ..., n$, is defined as a couple (V_{C_i}, V_{lts}), where V_{C_i} is a set of vectors for components C_i, and V_{lts} is a vector LTS that indicates the order of interactions of the vectors V_{C_i}.

Reordering of messages is needed in some communication scenarios to ensure a correct interaction when two communicating entities have messages which are not ordered as required. In our proposal, such a reordering of messages can be specified making it explicit in the writing of the adaptation contract.

Example. As we checked in Section 5, there exist mismatches in the component interaction. Therefore, an adaptation contract is used to solve these problems. The adaptation contract is specified by a set of synchronous vectors, and a vector LTS. Synchronous vectors are automatically obtained from the matching tuple sets ($MT_{CR,R}$ and $MT_{R,RD}$) calculated in Section 4, which indicate the best matching between the interfaces corresponding to the Client-restaurant and the composite component Restaurant and RestDB. The generation of the vector LTS is assisted according to the synchronous vectors, and it specifies the ordering of execution of the vectors in order to generate the adaptor. As we consider value-passing protocol and data exchanged by means of messages, we need to resolve conflicts at this level. To do that we relate the parameters of the messages binding parameter names when required. Considering the Client-restaurant request, we need to bind the parameters *list* in *CR* with *VIPlist* and *Guestlist* in *R*. Thus, we will have, for instance, *getList?list* in *CR* related to *[priv==“VIP”]listRest?list* or *[priv==“Guest”]listRest?list* in *R*. The adaptation contract related to the interaction between *CR* and the composite component *R* and *RD* is composed by the vectors presented below (labels l_{cr_1}, l_{r_1}, ..., are represented in Figure 5), and the vector LTS depicted in Figure 8.

$v_1 = \langle l_{cr_1}, l_{r_1} \rangle$, $v_2 = \langle l_{cr_2}, l_{r_2} \rangle$, $v_3 = \langle l_{cr_2}, l_{r_3} \rangle$, $v_4 = \langle l_{cr_3}, l_{r_4} \rangle$, $v_5 = \langle l_{cr_4}, l_{r_7} \rangle$,
$v_6 = \langle l_{cr_5}, l_{r_8} \rangle$, $v_7 = \langle l_{cr_6}, l_{r_9} \rangle$, $v_8 = \langle l_{r_5}, l_{rd_1} \rangle$, $v_9 = \langle l_{r_6}, l_{rd_2} \rangle$

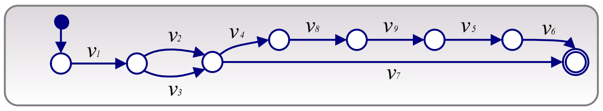

Figure 8: Vector LTS indicating the ordering of the interaction among the Client-restaurant request and the composite component Restaurant and RestDB

For the Client-taxi request CT we can obtain the corresponding adaptation contract following the same procedure.

Finally, we generate a correct CA-STS adaptor. We have to transform this adaptor in a WF adaptor component to complete our process.

6.2 Adaptor Generation

We generate an adaptor specification from a set of interfaces (client and components), and an adaptation contract. The adaptor acts as a third-party component that is in charge of coordinating the client and all the components involved in the system *w.r.t.* a set of interactions defined in the contract. Thus, at this stage, a CA-STS adaptor can be generated by using the tool `(D)Compositor` described in [Mateescu et al. 2008] (incorporated into the toolbox `ITACA`), and conditions and contexts are considered. This algorithm uses CADP [Garavel et al. 2007] to get an adaptor where all the possible deadlocks are removed.

Example. Figure 9 shows the CA-STS adaptor for the interaction between Client-restaurant CR and the composite component Restaurant-RestDB R-RD.

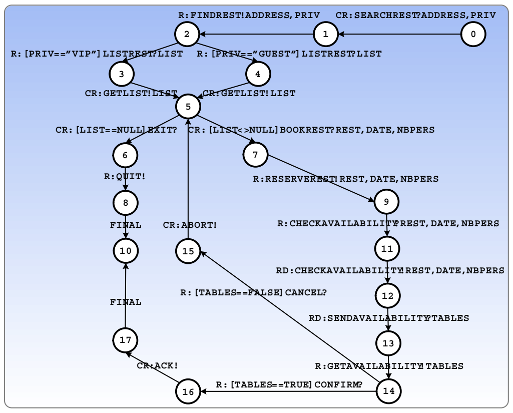
R:FINDREST!ADDRESS,PRIV
CR:SEARCHREST?ADDRESS,PRIV
0
1
2
R:[PRIV=="VIP"]LISTREST?LIST
R:[PRIV=="GUEST"]LISTREST?LIST
3
4
CR:GETLIST!LIST
CR:GETLIST!LIST
5
CR:[LIST==NULL]EXIT?
CR:[LIST<>NULL]BOOKREST?REST,DATE,NBPERS
6
7
R:QUIT!
R:RESERVEREST!REST,DATE,NBPERS
8
9
FINAL
CR:ABORT!
R:CHECKAVAILABILITY?REST,DATE,NBPERS
10
15
11
RD:CHECKAVAILABILITY!REST,DATE,NBPERS
R:[TABLES==FALSE]CANCEL?
12
FINAL
RD:SENDAVAILABILITY?TABLES
17
13
CR:ACK!
R:GETAVAILABILITY!TABLES
R:[TABLES==TRUE]CONFIRM?
16
14

Figure 9: Adaptor specification for the interaction between CR with R-RD

Component labels are prefixed by the component identifier, namely CR, R, and RD to uniquely identify all the labels involved in the system. Note message directions are reversed because all messages corresponding to the Client-restaurant request and the Restaurant and RestDB components will go through the adaptor, and this latter has to synchronize with these messages using complementary directions. The set of properties defined in Section 5 are preserved in the composition, so the CA-STS adaptor is free of inconsistences. It is worth mentioning that in case a problem occurs during this component composition, such as exception or connection loss, our adaptation process will find for new components that reply the client request, by generating a new adaptor on-the-fly for the remaining interactions. In such a way, the client request can be executed even when some interaction problems are arisen. We can generate a CA-STS adaptor for the Client-taxi request following the previous process.

6.3 Obtaining WF Adaptor from CA-STS Adaptor

We have to generate a WF component from the CA-STS adaptor by using our model transformation process. This WF component constitutes the WF adaptor, which will be deployed to allow the correct connection of client and components.

Our interface model (CA-STS specification) can take into account some additional behaviours (interleavings) that cannot be implemented into executable languages (*e.g.*, WF or BPEL). In order to make platform-independent adaptor implementable *wrt.* a specific platform, some filters are used with the purpose of pruning parts of the CA-STS corresponding to these interleavings and keep only executable paths. In particular, to implement an adaptor interface model (CA-STS adaptor) as an adaptor component (WF adaptor), we proceed in three steps: (i) filtering the interleaving cases that cannot be implemented (*e.g.*, several emissions and receptions outgoing from a same state), (ii) checking that the pruning does not affect the correct functionality, and (iii) encoding the filtered model into the corresponding implementation language (in our case WF) using our transformation process. Next, we give some guidelines for this encoding.

First, the initial state of the CA-TS is encoded as the initial state of the WF workflow. Final states are encoded as `Terminate` activities.

The transformation process derives step by step parts of the abstract workflow by focusing on one state of the CA-STS after the other. We distinguish in the following the translation of transitions corresponding to message activities (`Receive`, `Send`), and the generation of structuring activities (`Sequence`, `While`, `IfElse`, `Listen`). Let us start with messages:

– a transition with one reception as labels is translated into a `Receive` activity;
– a transition with one emission corresponds to a `Send` activity.

Now, we focus on the encoding of the CA-STS structuring into the workflow:

– a `Sequence` activity is generated for a sequence of transitions in the CA-STS corresponding to two successive message activities, and for which no states involve more than one outgoing transition;
– a cycle in an CA-STS is translated using a `While` activity. If several cycles loop on a same state, it corresponds to a single `While` activity. However if a cycle in the CA-STS contains another (local) cycle, this latter will also be translated as a `While` activity nested in the outmost one.

– if the state of the CA-STS to be translated involves two or more outgoing transitions:
  - if all the outgoing transitions hold inputs, a `Listen` activity is derived,
  - otherwise a conditional choice `IfElse` activity is generated.

Finally, all the needed pieces of $C\#$ code will be added by hand while refining the abstract workflow into a real WF workflow. We consider that a `While` construct always has priority over `IfElse` and `Listen`. Thus, when our process finds a pattern which can be transformed into `While` and `Listen` at the same time, we first will apply the `While` pattern and secondly the `Listen` one. The prototype tool that computes the transformation to implement the adaptor, is called `CASTS2WF`. It considers the guideline described previously and uses a state machine pattern based on the transformation process from CA-STS protocol elements to WF workflow activities presented in Figure 3 and here described.
Example. Figure 10 gives the WF workflow obtained from our CA-STS adaptor. The same procedure is performed to transform the CA-STS adaptor of the Client-taxi request in a new WF workflow (not presented here for space reasons).

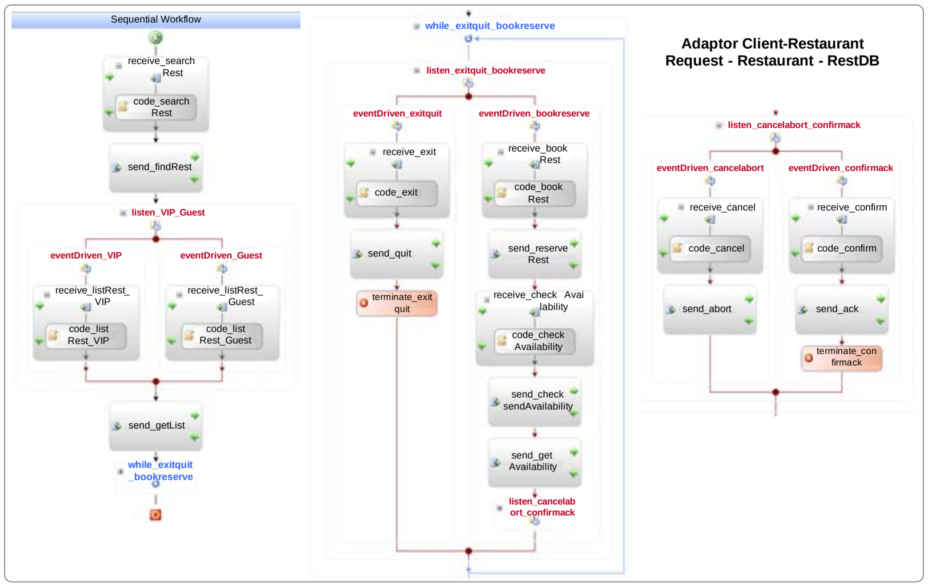

Figure 10: WF workflow corresponding to the Adaptor component interface that communicate CR with the composite component R with RD

7 Evaluation of Experimental Results and Discussion

The different steps of our composition and adaptation approach presented in this paper, have been implemented in a set of tools that constitute a framework called `DAMASCo` (Discovery, Adaptation and Monitoring of Context-Aware Services and

Components), which is integrated in our toolbox ITACA. In order to evaluate the benefits of our approach to find out the best component interfaces for a concrete request and to generate an adaptation contract in terms of development effort required and accuracy of the contract generated, DAMASCo has been validated on several examples, such as an on-line computer material store, a travel agency, a road info system or the case study presented here: an on-line booking system. These scenarios have been implemented in the WF platform by us and executed on an Intel Pentium(R)D CPU 3GHz, 3GB RAM computer. This represents an initial stage, checking our whole framework, but the main goal of our approach is to support industrial systems by validating directly pre-existing applications in the real-world. Table 2 shows the experimental results (CPU load and execution time) corresponding to the example presented throughout this article, as well as a road info system. For each problem, we have studied three different scenarios, which are organised according to increasing size and complexity *w.r.t.* the number of interfaces involved, as well as the overall size of protocols as a total number of states and transitions.

Problem	Scenario	Size			Parameter	
	(Problem Version)	Interfaces	States	Transitions	CPU(%)	Time(s)
ebooking	eb-v004	4	28	33	11,1	0,110
	eb-v005	25	128	160	16,2	0,688
	eb-v007	67	352	440	34,1	1,719
roadinfo	ri-v006	6	32	36	13,8	0,249
	ri-v008	44	264	302	20,7	0,954
	ri-v010	103	650	780	47,6	2,437

Table 2: Evaluation results of our scenario

For example, for the on-line booking system (ebooking), we have checked the discovery process for the Client-restaurant request with a component repository containing (i) the three interfaces corresponding to the Restaurant, RestDB and Taxi Components presented in Fig. 5 (eb-v004), (ii) those three interfaces plus 21 other interfaces related to flights, hotels, cinemas, theaters, and so on (eb-v005), and (iii) the three interfaces plus 63 more protocols including some dummy interfaces to check the scalability of our framework (eb-v007). These two latter tests have been performed to illustrate that the complexity of the problem does not seriously affect the effort required and the accuracy of our process as can be appreciated in the results shown in Table 2. In fact, the execution times show a linear dependency related to the number of transitions. Therefore, our approach turned out to be cost-effective, and with respect to the concepts compared and execution time, it was more efficient than the mechanism used in the WordNet::Similarity package[14]. In addition, using this package, only English terms defined in its data base can be compared. In our approach, one can define a specific domain ontology and use it without any restrictions comparing ontology

[14] http://wn-similarity.sourceforge.net/ Accessed on 20 September 2010.

concepts by means of matching patterns.

Nevertheless, our approach has some limitations. Thus, although ontologies represent certain advantages to discover components, and even our approach could adopt other mechanisms different from ontologies, our process depends on the construction of shared domain ontologies, and there is no component market existing today which produces ontologies in an standard way. Another limitation of our approach is to do with using the "average" for ranking the components selected in the discovery process. On the other hand, since our proposal supports synchronous systems with a client/server model, we adopt a synchronous and binary model, so we can not simulate asynchronous systems with our model.

8 Related Work

We compare our approach with related works in software composition and adaptation, especially those which focus on reusing components and on tackling the interoperability issues which exist at the different levels of component interaction. We also present works based on model transformation, thereby relating their approaches to existing programming languages and platforms.

Several proposals [Gaspari and Zavattaro 1999, Schmidt and Reussner 2002], [Inverardi and Tivoli 2003, Cámara et al. 2008, Canal et al. 2006b] focus on the signature and behavioural levels, and advocates abstract notations (*e.g.*, message correspondences or vector regular expressions) and algorithms to generate adaptor protocols. Gaspari and Zavattaro [Gaspari and Zavattaro 1999] study the operational behaviour of the CORBA Messaging Service from different perspectives in order to facilitate the task of implementors.

In [Schmidt and Reussner 2002] the authors present an adaptation approach as a solution to particular synchronisation problems between concurrent components Inverardi and Tivoli [Inverardi and Tivoli 2003] tackle the automatic synthesis of connectors in COM/DCOM environments, by guaranteeing deadlock-free interactions among components. They may also define properties that the resulting system should verify using liveness and safety properties expressed as specific processes. Compared to these proposals, we may match different name messages using the correspondences of our adaptation contract, which is very useful within context-aware systems. In addition, these approaches do not use any mapping language for the adaptor specification, so the adaptor is restricted to possible non-deadlocking behaviours [Inverardi and Tivoli 2003]. Nevertheless, the rich notation we have proposed allows us to deal with possibly realistic and complex adaptation scenarios and it is possible to address behavioural adaptation. In [Canal et al. 2006b] a solution to behavioural adaptations is proposed using regular expressions of vectors as a mapping notation. This work is supported by algorithms based on synchronous products and Petri nets encodings. Our mapping notation is as expressive as regular expressions of vectors since we use synchronisation vectors, as well as transition systems that can express the sequence, choice and iteration operators of regular expressions. Besides, thanks to our automaton-based notation, we tackle the complexity of the mapping using a divide-and-conquer approach, which makes its writing easier. In [Cámara et al. 2008] the authors present a composition and adaptation approach based on transition systems. However, their approach only considers the signature and behavioural levels to generate adaptors, and abstracts from the implementation frameworks.

Context-aware computing is concerned about the design and implementation of applications which are able to modify their functionality depending on changing conditions of the environment and the user. Many authors have studied context-aware computing, and have built pervasive applications [Autili et al. 2009, Marconi et al. 2009, Mokhtar et al. 2006, Schilit et al. 1994] to demonstrate the usefulness of this technology. There have even been significant achievements in the architectural support of context-aware applications, such as [Chen et al. 2003, Salber et al. 1999]. However, at the different interoperability levels, the composition and adaptation of software entities within the pervasive systems has only briefly been dealt with in some of these works. As an example, an interesting proposal in this field is that of Ben Mokhtar *et al.* [Mokhtar et al. 2006]. They model OWL-S processes, namely both client's request and services, as finite state automata considering contexts. Nevertheless, their approach does not consider the behavioural compatibility, so deadlock-freeness cannot be checked. Related to context-aware adaptation, Autili *et al.* [Autili et al. 2009] present an approach to context-aware adaptive services. Services are implemented as adaptable components by using the CHAMELEON framework. This approach considers context information at design time, but the context changes at run-time are not evaluated. We consider context changes not only at design-time, but also at run-time.

As regards semantic-based composition, Brogi *et al.* [Brogi et al. 2008] base on the hypergraph theory to discover parts of descriptions of DAML-S/OWL-S services capable of satisfying a query when no single service can satisfy it, though they do not take the dynamic nature of the context information into account. With respect to the relationship between existing programming languages and platforms, the work presented by Brogi and Popescu [Brogi and Popescu 2006] outlines a methodology for the automated generation of adaptors capable of solving behavioural mismatches between BPEL processes. Compared to this work, our adaptation approach is able to reorder messages in between components when required, since our discovery process that generates the contract allows this facility. Finally, in [Motahari et al. 2007] the authors present techniques based on SCA components, by providing semi-automated support for identification and resolution of mismatches. We generate WF adaptors that consider not only signature and protocol mismatches, but also context and semantic information.

9 Concluding Remarks

In this article, we have presented our proposal related to context-aware composition and adaptation of software components. It automatically generates an adaptor when it is required, because when developing systems by reusing components, compositional issues are raised, so most components cannot be directly reused. To tackle these issues, we use model transformation. First, WF components of a system are transformed into their corresponding CA-STS specifications. These specifications are defined as transition systems, since these systems provide an expressive and graphical notation that specifies flexible adaptation policies between the interfaces of entities to be integrated. Then, we perform verification techniques to identify mismatch situations that will determine whether the components need adaptation or not, and to validate the components amongst a set of properties by applying symbolic model checking. Next, if adaptation is required, then we automatically generate a CA-STS adaptor from an adaptation contract and the CA-STS specifications. Finally, a WF adaptor component, which is deployed with the whole system, is extracted from the CA-STS adaptor.

This work aims at demonstrating that software composition and adaptation can be of real interest for widely used implementation platforms such as WF (.NET), and can help the developer when building software applications by reusing software components. The formal foundations of the different steps of our proposal have been implemented in a set of prototype tools constituting the framework `DAMASCo`, which has been validated in several examples. Putting the implementation into practise, we have compared criteria on the suitability of both platforms, WF and BPEL. Thus, we have also carried out experiments on the implementation of adaptors using BPEL and the Netbeans Enterprise.

As regards plans for future work, we intend to extend our proposal to tackle dynamic reconfiguration of components, by handling the addition or elimination of both components and context information. A first approach has been published in [Cansado et al. 2009]. We also plan to design our model transformation process as a metamodel underlying on MDA, as well as to perform formal demonstrations to determine the correction and completeness of this process. Another perspective is to study the use of more powerful techniques, such as Multi Criteria Decision Making Methods (MCDM), to rank the components discovered, instead of the "average" of the degrees of match. Finally, another line of future work is to define context ontologies to be used in our proposal, with the purpose of making explicit context information for each concept of an ontology.

Acknowledgements

This work is partially supported by the projects TIN2008-05932, and P06-TIC-02250 and P07-TIC-03131, funded by the Spanish Ministry MICINN and FEDER, and the Andalusian local Government, respectively. The authors are grateful to the anonymous referees who helped to improve the contents and quality of this article.

References

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

[Autili et al. 2009] M. Autili and P. Di Benedetto and P. Inverardi. Context-Aware Adaptive Services: The PLASTIC Approach . In *Proc. of FASE'09*, volume 5503 of *LNCS*, pages 124–139. Springer, 2009.

[Bordeaux et al. 2004] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, volume 3324 of *LNCS*, pages 15–28. Springer, 2004.

[Brogi et al. 2008] A. Brogi and S. Corfini and R. Popescu. Semantics-based composition-oriented discovery of Web services. *ACM Transactions on Internet Technology*, 8(4):19:1–19:39, 2008.

[Brogi and Popescu 2006] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proc. of ICSOC'06*, volume 4294 of *LNCS*, pages 63–77. Springer, 2006.

[Bryant 1986] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bukovics 2008] B. Bukovics. *Pro WF: Windows Workflow in .NET 3.5*. APress, 2008.

[Burton-Jones et al. 2003] A. Burton-Jones, V.C. Storey, V. Sugumaran, and S. Purao. A Heuristic-Based Methodology for Semantic Augmentation of User Queries on the Web. In *Proc. of ER'03*, volume 2813 of *LNCS*, pages 476–489. Springer, 2003.

[Cámara et al. 2009] 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*, pages 627–630. IEEE CS, 2009.

[Cámara et al. 2008] J. Cámara and G. Salaün and C. Canal. Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *Journal of Universal Computer Science*, 14(13):2182–2211, 2008.

[Canal et al. 2006a] C. Canal, J.M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities.

[Canal et al. 2006b] C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *Proc. of FMOODS'06*, volume 4037 of *LNCS*, pages 63–77. Springer, 2006.

[Cansado et al. 2009] A. Cansado, C. Canal, G. Salaün, and J. Cubo. A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. In *Proc. of FACS'09*, volume 263 of *ENTCS*, pages 95–110. Elsevier, 2010.

[Chen et al. 2003] H. Chen, T. Finin, and A. Joshi. An Intelligent Broker for Context-Aware Systems. In *Proc. of UbiComp'03*, volume 2864 of *LNCS*, pages 183–184. Springer, 2003.

[Cubo et al. 2009a] J. Cubo, C. Canal, E. Pimentel, and G. Salaün. A Formal Model and Composition Language for Context-Aware Service Protocols. In *Proc. of CASTA'09*, pages 17–20. ACM Digital Library, 2009.

[Cubo et al. 2007a] J. Cubo, G. Salaün, J. Cámara, C. Canal, and E. Pimentel. Context-Based Adaptation of Component Behavioural Interfaces. In *Proc. of COORDINATION'07*, volume 4467 of *LNCS*, pages 305–323. Springer, 2007.

[Cubo et al. 2009b] J. Cubo, M. Sama, F. Raimondi, and D.S. Rosenblum. A Model to Design and Verify Context-Aware Adaptive Service Composition. In *Proc. of SCC'09*, pages 184–191. IEEE CS, 2009.

[Dey and Abowd 2000] A.K. Dey and G.D. 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*, pages 304–307, 2000.

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

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

[Garavel et al. 2007] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 158–163. Springer, 2007.

[Gaspari and Zavattaro 1999] M. Gaspari and G. Zavattaro. A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service. In *Proc. of ECOOP'99*, volume 1628 of *LNCS*, pages 495–518. Springer 1999.

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

[Marconi et al. 2009] A. Marconi and M. Pistore and A. Sirbu and H. Eberle and F. Leymann and T. Unger. Enabling Adaptation of Pervasive Flows: Built-in Contextual Adaptation. In *Proc. of ICSOC-ServiceWave'09*, volume 5900 of *LNCS*, pages 445–454. Springer, 2009.

[Mateescu et al. 2008] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.

[Mokhtar et al. 2006] S.B. Mokhtar, D. Fournier, N. Georgantas, and V. Issarny. Context-Aware Service Composition in Pervasive Computing Environments. In *Proc. of RISE'05*, volume 3943 of *LNCS*, pages 129–144. Springer, 2006.

[Motahari et al. 2007] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-Automated Adaptation of Service Interactions. In *Proc. of WWW'07*. ACM Press, 2007.

[Paolucci et al. 2002] M. Paolucci, T. Kawamura, T.R. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proc. of ISWC'02*, volume 2342 of *LNCS*, pages 333–347. Springer, 2002.

[Patil et al. 2004] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. METEOR-S Web Service Annotation Framework. In *Proc. of WWW'04*, pages 553–562. ACM Press, 2004.

[Salber et al. 1999] D. Salber, A.K. Dey, and G.D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *Proc. of CHI'99*, pages 434–441. ACM Press, 1999.

[Schilit et al. 1994] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *Proc. of WMCSA'94*, pages 85–90. IEEE CS, 1994.

[Schmidt and Reussner 2002] H.W. Schmidt and R.H. Reussner. Generating Adapters For Concurrent Component Protocol Synchronisation. In *Proc. of FMOODS'02*, pages 213–229. Kluwer Academic Publishers, 2002.

[Mostéfaoui and Hirsbrunner 2003] S.K.. Mostéfaoui, and B. Hirsbrunner. Towards a Context-Based Service Composition Framework. In *Proc. of ICSW'03*, pages 42–45. CSREA Press, 2003.

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

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

[Zapletal 2008] M. Zapletal. Deriving Business Service Interfaces in Windows Workflow from UMM Transactions. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 498–504. Springer, 2008.

[Zapletal et al. 2009] M. Zapletal and W.M.P. van der Aalst and N. Russell and P. Liegl and H. Werthner. An Analysis of Windows Workflow's Control-Flow Expressiveness. In *Proc. of ECOWS'09*, pages 200–209. IEEE CS, 2009.