A Model to Design and Verify Context-Aware Adaptive Service Composition

Javier Cubo
Dept. of Computer Science
University of Málaga
Málaga, Spain
cubo@lcc.uma.es

Michele Sama, Franco Raimondi and David Rosenblum
Dept. of Computer Science
University College London
London, UK
{m.sama,f.raimondi,d.rosenblum}@cs.ucl.ac.uk

Abstract

The introduction of mobile clients and context-aware behaviours into Web Service compositions may generate faults and inconsistencies. We introduce an extension of a composition model where context-awareness is made explicit and a number of correctness properties are verifiable. In particular, our extended model enables the verification of properties commonly used to validate context dependent applications. We also propose a set of algorithms to verify these properties efficiently.

1 Introduction

Web-Services composition is one of the most powerful instruments of the Service Oriented Architecture (SOA) allowing third parties to sell their services and services developers to reuse existing ones. In a classic web environment, in which statically all the requests are served in the same way, the service composition is straightforward. The introduction of web-enabled hand-held devices has created the necessity of a more context oriented composition in which the produced response is aware of certain contextual information on the requesting client.

Browsers running on hand-held devices transmit contextual information by adding them in the header of their HTTP requests. They usually do that implicitly inferring certain information directly from the device and encoding them in a custom format which may not be interpreted correctly by the invoked service. For instance consider a request to the inbox of Google Mail [13] performed on a Nokia N95 using two different browsers: Safari and OperaMini. The backend of Google Mail tries to contextualise the response for the requesting but client misunderstands the request from OperaMini and responds by transmitting the default frontend, which is too large to be rendered on a mobile phone and requires more time to be received due to its larger size.

The requests performed by Safari and OperaMini only differs in their headers, in which both browsers have included contextual information, with the idea that the Web Service would use them correctly.

If a Web Service uses the HTTP header to contextualise its response, it needs to know which parameters are being used, or it may send the wrong response, as it happens for Google Mail. Web Services can easily access contextual information in their implementation but this creates an implicit binding between their response and the context. This binding is implicit because in the service descriptor there is no trace of such dependencies. The problem is exacerbated when context-dependent Web Services are composed, because their composition has no explicit control on contextual information sent among the composed services. Moreover this introduces complexity both in the implementation of the single service (which should be adaptive) and in their composition because, in order for the response to be correct, all the services should have adapted in a consistent way.

Generally, this problem has been avoided by replicating contextual information as get/post parameters, thus increasing the size of each request. Even disregarding the issue of this overhead, replicating contextual information as parameters introduces the problem of naming them. Indeed, different acronyms or abbreviations are likely to be used for the same context variable when combining Web Services from different producers, and their composition may end up containing duplicated parameters. For instance, the following is the full address line for a search on the HP website:

```
http://search.hp.com/query.html?lang=en
 &submit.x=0&submit.y=0&qt=query&la=en&cc=us
```

Note that the language is specified twice with two different names: *'lang'* and *'la'*. Also note that the real contextual language for this request was *'en_gb'*, while the *'cc'* (country code) parameter specifies *'us'* instead of *'gb'*, which is wrongly inferred somehow. Moreover if multiple services are contextualising their response using different contexts, for instance using the keyboard layout or the browser language, the response may be inconsistent. It can also happen that two services are requiring two different parameters but they are calling them with the same name. Their composition needs to distinguish them by introducing an extra layer

978-0-7695-3811-2/09 $26.00 © 2009 IEEE
DOI 10.1109/SCC.2009.61

of complexity and overhead to avoid inconsistencies.

In this paper we address the problem of making Web Services and their composition context-aware. In Section 2 we propose a definition model for context-aware Web Services. It is an extension of our previous model [6] in which we include context-aware features for Web Services composition. Section 3 presents a set of validation patterns and verification algorithms to check context-aware services. In Section 4, we discuss mismatch and inconsistence problems relative to context-aware service composition, as well as a proposal for solving those problems by generating a third-party service (*adaptor*) that fits the service interaction correctly. In Section 5, we discuss the benefit of using this model in terms of composition and testability with respect to some related works. Finally, Section 6 concludes the paper and sketches future works.

2 Modeling Context-Aware Service Protocols

This section presents a model to formalise context-aware service composition. Intuitively, our idea is to extend services descriptors including the HTTP header's parameters required by the services to use contextual information. Our model is built on top of a stack of standard protocols and enriches their capabilities with a context-aware semantic, since the original stack is not context-aware.

To illustrate our approach, we introduce a simplified scenario which we use as a case study through the paper.

2.1 Running Example

Consider a *Client* who requests maps from a *Map Service*, in street or satellite view mode. The *Map Service* uses the composition of *Street Service* and *Satellite Service*, which are implemented by separated third parties.

When the Client invokes *Map Service* for the first time, it may or may not provide its initial location as part of its header. If it is not provided, *Map Service* requests it explicitly. Map Service invokes *Street Service* and *Satellite Service* with an appropriate header containing a set of contextual information. *Street Service* uses the language provided by the invoking browser to localise the response, while *Satellite Service* uses the keyboard encoding. Furthermore, the *Street Service* supports any specified language in the browser, while the *Satellite Service* only supports *'en_gb'* and *'en_us'*, so if the keyboard encoding language is different from these ones, it will reply with a default one.

The composition of such services must be able to contextualise to the Client configuration responding in language consistent in the composition and suitable for the Client.

2.2 Behavioural Model

We assume that services are specified using both a signature and a protocol. We build upon the SOA stack [7]. Signatures correspond to operation profiles specified using WSDL, and protocols are business processes defined in industrial platforms, such as BPEL [1] or WF workflows [21]. In the proposed model, protocols, which may be instantiated to communicate with other different protocols, are represented by means of Labelled Transition Systems (LTSs) extended with value passing [18], contexts and conditions, called *Context-Aware Symbolic Transition System* (CA-STS). The contextual information can be inferred from the header block of SOAP message (or from entities such as sensors or devices), and it may also be added to the WSDL description [15].

Definition 1 (Context) *A Context is a set of couples* (CA, CV) *where:* CA *is a context attribute (e.g., lang), and* CV *is the value of* CA*, which can be a single value (e.g., ['en_gb']) or a set of values ((e.g., ['en_gb', 'es_es'])).*

Definition 2 (CA-STS label) *A label corresponding to a transition of a CA-STS is either an internal action or a tuple* (C, SI, M, D, PL, CL) *where:* C *are the conditions of the message (represented by a boolean expression),* SI *is a service identifier,* M *is the message name,* D *is the direction of messages*[1]*,* PL *is either a list of data terms for emission or of variables for reception (which may include contexts given explicitly by the user), and* CL *is a list of context values for emission or of context attributes for reception (implicit contexts - Def. 1 - inferred from the HTTP header of the message). Condition's expressions and contexts are prefixed by service identifiers (e.g., [c:lang == 'en_gb']).*

Definition 3 (CA-STS) *A Context-Aware Symbolic Transition System (CA-STS) is a tuple* (A, S, I, F, T) *where:* A *is an alphabet of messages (represented by CA-STS labels),* S *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are final states, and* $T \subseteq S \times A \times S$ *is a transition function.*

We need to match conditions, data parameters, as well as message contexts of services interacting. We use a synchronous communication model which may be: (i) 1-ary, (ii) binary between emission/reception messages, or (iii) n-ary among a sender and more than one receiver (broadcast communication). Note we do not apply a close world assumption. For us only predicates which are explicitly stated as True or False can be used. Condition and contexts should be fixed for all the services interacting in the system. So far, we control this by prefixing condition's expressions and contexts with their service identifiers. We plan to extend the model to solve this using a *context stack* in the contextual mapping, and the introduction of constraints on this stack.

CA-STS services for our example. Figure 1 depicts the scenario described in Section 2.1 by using the formalism introduced in this section. Initial and final states are marked using bullet arrows and darkened states, respectively (notice

[1] We use the standard notation in which *!* and *?* represent emission and reception respectively.

the use of service identifiers, where c is for Client, m for Map, t for Street and s for Satellite services).

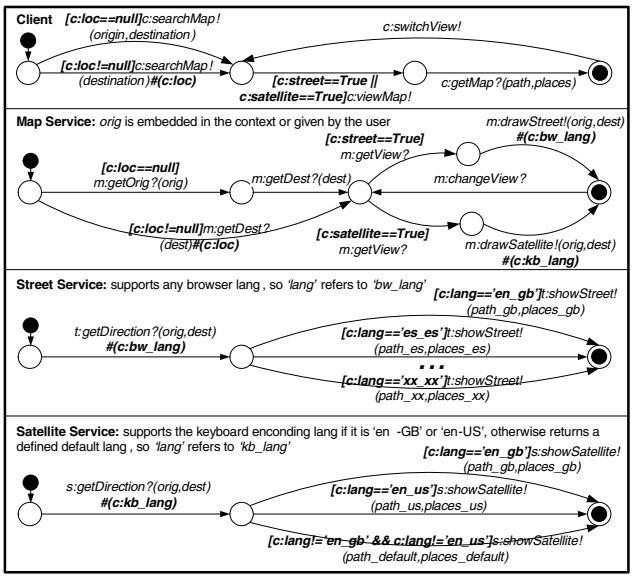

Figure 1. CA-STS behavioural model for the tourist map system.

The main contributions of this model are: (1) *message conditions* and (2) *context attributes* which can be derived automatically from HTTP headers. As an example, consider the condition $(c : loc! = null)$ in the Client service in Figure 1: this condition means the client location is known (because it can be inferred directly from the HTTP header). However, if the location is not known, then the client has to send it like an explicit parameter.

3 Verification Model

Before performing the process of service composition, we need to validate each CA-STS service to verify that they are free of faults and inconsistences. Our verification model consists of a set of validation patterns and their corresponding verification algorithms based on Ordered Binary Decision Diagrams (OBDDs [3]), which are presented below.

3.1 Validation Patterns

Once context-awareness has been introduced into the service model we can validate it against a set of properties:

- Determinism: in each state in which the computation can follow different paths, conditions on those multiple requests/responses must be mutually exclusive.
- State liveness: if in a state contexts are used to select the next request/response, at least one combination of values of those contexts must lead to a transition.
- Request/response liveness: if a request/response is conditioned by a certain value of the context, that condition must be satisfiable.
- Non-blocking states: irrespective of the values of the contextual variables, communication should always reach a final state.

These properties are an extension of the ones presented by Sama *et al.* [20], which includes a more detailed explanation, but applied to CA-STS instead of reactive systems. The communication between client and server is supposed to be a finite flow of requests/responses in which context information are used to improve the provided service. Determinism is required to guarantee a correct mapping between context and transitions. If two conditions would be satisfiable simultaneously then the result would be nondeterministic and the result will depend on the implementation and not on the context itself. State liveness requires that at least one outbound transition is enabled for each state (with the exception of the final state). Request/response liveness guarantees that all the specified transitions will be satisfiable for at least one combination of contextual values. A condition like *[lang=='en_gb' && lang=='es_es']* will never be satisfied and the corresponding transition will never be executed. 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. In terms of context mapping all the possible combinations of context must satisfy one and only one transition. The properties presented above can be verified both for the CA-STS services and for the CA-STS adaptor generated in the composition (Section 4). In the next section we describe efficient algorithms for their verification.

3.2 Verification Algorithms

We have implemented a set of OBDD-based algorithms to verify the properties described in the previous section, and we describe an implementation of the algorithms.

3.2.1 OBDD Representation

Before describing the algorithms, we first describe how a CA-STS service can be represented by mean of OBDDs, a technique used in symbolic model checking [5]. 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 means of Boolean formulae. These formulae are then manipulated using OBDDs. In particular, we show how algorithms for the verification of the properties presented in Section 3.1 can be derived from our OBDD-based representation. Due to space limitations, we refer to [3, 5] for further details on OBDDs; for the scope of this paper, it is sufficient to present the Boolean encoding of the model and note the Boolean formulae can be manipulated by means of OBDDs.

As described in Section 2.2, let (A, S, I, F, T) be a CA-STS. The number n of Boolean variables required to encode the set of states S is $n = \lceil log_2|S| \rceil$. For instance, 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 or 1, with the assumption that the value 0 corresponds to the negation of a variable. 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[2]. 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 $a = (c, si, m, d, pl, cl) \in A = (C, SI, M, D, PL, CL)$. Message a is encoded as a Boolean formula by associating a Boolean variable to each parameter in pl and to each context variable in cl, and then representing the condition c by means of these variables. We denote with $\overline{a}$ the Boolean encoding of a given message $a \in A$ (essentially, this is the Boolean encoding of c together with the actual name of the message).

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, 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.,* $\overline{T} = \bigvee_{(s,a,t) \in T} (\overline{s} \wedge \overline{a} \wedge \overline{t})$.

The encoding presented above allows for the definition of the verification algorithms for the properties presented in Section 3.1. First, we compute the set of reachable states by means of Algorithm 1, where the reachable states are computed starting from the initial state I and by adding the set of states reachable from here iteratively until no change occurs (notice that at line 11 "primed" variables encoding successor states are converted back to "standard" variables before being added to the current set of reachable states).

The set of reachable states is used in the following algorithms to verify properties of a (single) CA-STS:

Determinism: Algorithm 2 computes the set of conditions (in the form of a BDD) that enable transitions to more than a destination state (notice: in our encoding we assume that the message name has been encoded in the condition itself). For each state (line 2) and for each condition available in that state (line 4), the set of states reachable is computed (line 5 and 6). If this set contains more than one state, the condition is added to the set of faulty conditions (line 8). Another kind of non-deterministic transition may occur: this is when two different conditions between the same pair of states can be satisfied at the same time. Detection of this non-deterministic error is performed by means of Algorithm 3 (notice that this algorithm is not symbolic and operates on the explicit set of transitions. This is because redundancies in disjunction of Boolean formulae are eliminated if using BDD). The function "between" at line 5 returns the collection of BDDs corresponding to the conditions of transitions between two fixed states s_i and s_j. At line 8 we verify whether the pairwise conjunction of these conditions is not equivalent to false. If this is the case, then two conditions can be true at the same time, leading to non-determinism in transition selection.

State liveness: Algorithm 4 computes the set of states from which the set of reachable states is empty (*i.e.,* is equivalent to false). If that happens the part of Web Service represented by the unreachable state is dead code.

Request/response liveness: Algorithm 5 simply checks whether there exists an assignment to the contextual variables such that a condition is equivalent to false. If that happens the faulty request/response is unreachable.

Non-blocking states: Algorithm 6 detects the set of non-

Algorithm 1 Reachable States

Input: the CA-STS encoded using OBDDs.
Output: *reach*: reachable states (OBDD).

```
1: BDD q,reach,next;
2: q = I;
3: reach = bddZero();
4: next = bddZero();
5: while q != reach do
6:     reach = q;
7:     next = q;
8:     next = next * T;
9:     next = exists(s,next);
10:    next = exists(a,next);
11:    next = next.swapVariables(s, s');
12:    q = q + next;
13: end while
14: return reach
```

Algorithm 2 Non-determinism detection

Input: the CA-STS encoded using OBDDs.
Output: *faults* (OBDD).

```
1: BDD conditions, next
2: for each state v ∈ S do
3:     conditions = exists(v',T ∧ v);
4:     for each condition c ∈ conditions do
5:         next = c ∧ T ∧ v
6:         next = exists({c, v},next);
7:         if size(next > 1) then
8:             faults.add(c);
9:         end if
10:    end for
11: end for
12: return faults
```

[2]By slight abuse of notation, the same symbols $s_i (i \in \{1, \ldots, s\})$ are used to denote Boolean variables or their value in a vector, and atomic propositions in logical formulae.

Algorithm 3 Non-deterministic condition detection

Input: the CA-STS encoded using OBDDs.
Output: *faults* (OBDD).

```
1: vector<BDD> tmp
2: vector<Transition> transitions
3: for i = 0; i < |S| do
4:     for j = i; j < |S| do
5:         tmp = between(transitions,s_i , s_j );
6:         for n = 0; n < |tmp| do
7:             for m = n; m < | tmp | do
8:                 if size(tmp[n] ∧ tmp[n] != ⊥ ) then
9:                     faults.add(tmp[n]);
10:                end if
11:            end for
12:        end for
13:    end for
14: end for
15: return faults
```

Algorithm 4 State liveness detection

Input: the CA-STS encoded using OBDDs.
Output: *faults* (OBDD).

```
1: BDD conditions, tmp
2: for each state v̄ ∈ S do
3:     tmp = T̄ ∧ v̄
4:     tmp = exists({v̄, c̄},tmp);
5:     if tmp ≡ ⊥ then
6:         faults.add(v̄);
7:     end if
8: end for
9: return faults
```

final states where the sum of the conditions of the outgoing assignments is different from true. This means that there exists at least an assignment for which there is no enabled transition, so that the service can produce a deadlock.

3.2.2 Prototype implementation

We have implemented the algorithms presented in the previous section in a prototype tool which is available to download from `http://forge.cs.ucl.ac.uk/projects/casts/`. The tool includes a set of data structures to encode instances of our CA-STS model. These data structures are translated automatically into BDD, and we employ the GPL library CUDD [22] to manipulate BDDs.

Verification of the CA-STS services of the tourist map system. The implementation of the algorithms presented above has shown the following faults in the model of Figure 1, which were not obvious at a first analysis:

- Non-deterministic condition in Client and Map Service: this fault is caused because our CA-STS does not require that the two conditions *[c:street==True]* and *[c:satellite==True]* are mutually exclusive.
- Non-deterministic condition in Street Service: this fault is caused by the "catch-all" condition *[c:lang=='xx_xx']* which is also satisfied when the language is *[c:lang=='en_gb']* or *[c:lang=='es_es']*.
- Blocking states in Map Service: if neither *[c:satellite==True]* nor *[c:street==True]*, then the third state from the left is blocking.

Algorithm 5 Rule liveness detection

Input: the CA-STS encoded using OBDDs.
Output: *faults* (OBDD).

```
1: BDD conditions
2: for each condition c̄ ∈ conditions do
3:     if c̄ ≡ ⊥ then
4:         faults.add(c̄);
5:     end if
6: end for
7: return faults
```

Algorithm 6 Blocking states detection

Input: the CA-STS encoded using OBDDs.
Output: *faults* (OBDD).

```
1: BDD tmp
2: for each state v̄ ∈ S do
3:     tmp = v̄ ∧ T̄
4:     tmp = exists({v̄', c̄},tmp);
5:     if tmp != ⊤ and not_final(v̄) then
6:         faults.add(v̄);
7:     end if
8: end for
9: return faults
```

- Blocking states in Street and Satellite Services: if language is null, the second state from the left is blocking.

The verification of these properties required less than 1 second for all the CA-STS services. We employed up to 9 Boolean BDD variables to encode our scenario, corresponding to a model of size 2^9. Notice that the properties presented above could not be checked using a standard model checker, because of the introduction 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.

4 Context-Aware Service Composition

Once we have checked the CA-STS services of the system, we can compose them. But not always services fit each other, so we have to generate a third-party service, called *adaptor protocol* in order to solve behavioural mismatches arisen in the service interaction [17]. To obtain the CA-STS adaptor, we define a *contextual mapping* that avoids not only the mismatches, but also the inconsistences detected in Section 3.2. Last, we verify that the resulting composition is correct, by validating the CA-STS adaptor. Our aim is to guarantee:

1. variable matching: if a request and a response are to be coupled, the request needs to provide all the contextual variables which the response is requiring;
2. value matching: if a request and a response are to be coupled and context variables are used, the response needs to handle a super-set of the possible values that the context variables (sent by the request) can assume.

4.1 Contextual Mapping of CA-STS Services

To achieve the aforementioned matching, we may need to rename messages, to group more than one request/response event, and/or to rename parameters. Formally, we introduce the notion of *synchronisation vectors* to synchronize an event occurring among a set of services:

Definition 4 (Synchronisation Vector) *A synchronisation vector (or simply vector) for a set of services* $\{W_{s_i} = (A_i, S_i, I_i, F_i, T_i)\}_{i\in\{1,..,n\}}$, *is a vector of messages* $\langle m_1, \ldots, m_n\rangle$ *with* $m_i \in A_i \cup \{\varepsilon\}$ *(*ε *meaning that a service does not participate in a synchronization).*

To simplify the notation, we will remove ϵ messages from synchronisation vectors. 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 vectors. This order between vectors is essential in some situations in which mismatch can be avoided by applying some vectors in a specific order. We model the composition of services by introducing the notion of *contextual mapping*, that makes use of vectors.

Definition 5 (Contextual Mapping) *A contextual mapping (also called contract) for a set of services* W_{s_i}, $i \in 1, ..., n$, *is defined as a couple* (V_{W_s}, V_{lts}), *where* V_{W_s} *is a set of vectors for services* W_{s_i}, *and* V_{lts} *is a vector LTS.*

We need to check wether mismatches exist in the service interaction, as it may happen that the services of a scenario cannot be used together directly, and mismatches can occur at several levels: (i) message names (*e.g., c:switchView!* in Client versus *m:changeView?* in Map service), (ii) correspondences between several messages and a single one, as well as (iii) parameters mismatches (*e.g., [c:loc==null]c:searchMap!(origin,destination)* in Client with *[c:location=null]m:getOrig?(orig).m:getDest?(dest)* in Map service). Contextual mapping is used to solve these problems. As an example, consider the scenario described in 2.1 and the CA-STS depicted in Figure 1. To connect $[c : loc! = null]c : searchMap!(destination)\#(c : loc)$ with $[c : loc! = null]m : getDest?(dest)\#(c : loc)$ we introduce the synchronisation vector (notice the binding of some parameters such as *"destination"* to solve the mismatches of message names, as well as the renaming in the receptions of context variables by means of the ^ symbol):

$v_1 = \langle [c : loc! = null]c : searchMap!(dest)\#(c : loc),$
$[c : loc! = null]m : getDest?(dest)\#(c : loc\hat{}orig)\rangle$

Contextual Mapping for the tourist map system. The contextual mapping for our example scenario is specified by a set of synchronous vectors, and a vector LTS. To obtain the vectors we extend the automatic generation of [17], which looks for the most suitable contract solving the behavioural mismatches. We extend the generation process (considering conditions and contexts): (i) automatically checking each service against the properties defined in Section 3.1, by applying the algorithms described in Section 3.2, and (ii) manually with the designer intervention to solve the detected faults raised by conditions and contexts.

Therefore, in our contextual mapping, we need to solve both mismatch and inconsistence problems. First, we describe the solutions to take into account in our mapping, for the inconsistences detected in Section 3.2:

- Non-deterministic condition in Client and Map Service: this fault is solved by specifying that the conditions *[c:street==True]* and *[c:satellite==True]* are mutually exclusive by means of vectors v_4 and v_5.
- Non-deterministic condition in Street Service: this fault is solved by increasing the condition *[c:lang=='xx_xx']* with the restrictions *[c:lang=='en_gb']* $\wedge$ *[c:lang=='es_es']* in v_{10}.
- Blocking states in Map Service: we start the composition with the condition *[c:street==True]* by default.
- Blocking states in Street and Satellite Services: in this case we assume language is different to null.

Last, the contract generated, which solves both mismatch and inconsistence problems, is represented by both synchronisation vectors (below) and vector LTS (Figure 2). This latter indicates the ordering of execution of the vectors to generate the adaptor.

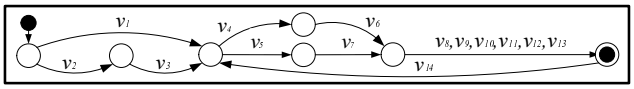

Figure 2. Vector LTS indicating the ordering of the interaction among the services.

$v_1 = \langle [c : loc! = null]c : searchMap!(dest)\#(c : loc),$
$[c : loc! = null]m : getDest?(dest)\#(c : loc\hat{}orig)\rangle$
$v_2 = \langle [c : loc == null]c : searchMap!(orig, dest),$
$[c : loc == null]m : getOrig?(orig)\rangle$
$v_3 = \langle m : getDest?(dest)\rangle$
$v_4 = \langle [c : street == True\&\&c : satellite == False]$
$c : viewMap!, [c : street == True]t : getView?\rangle$
$v_5 = \langle [c : street == False\&\&c : satellite == True]$
$c : viewMap!, [c : satellite == True]s : getView?\rangle$
$v_6 = \langle m : drawStreet!(orig, dest)\#(c : bw\_lang),$
$t : getDirection?(orig, dest)\#(c : bw\_lang\hat{}lang)\rangle$
$v_7 = \langle m : drawSatellite!(orig, dest)\#(c : kb\_lang),$
$s : getDirection?(orig, dest)\#(kb\_lang\hat{}lang)\rangle$
$v_8 = \langle c : getMap?(path, places),$
$[c : lang ='' en\_gb']t : showStreet!(path, places)\rangle$
$v_9 = \langle c : getMap?(path, places),$
$[c : lang ='' es\_es']t : showStreet!(path, places)\rangle$
$v_{10} = \langle c : getMap?(path, places),$
$[c : lang ='' xx\_xx'\&\&c : lang! ='' en\_gb'\&\&$

$c : lang! =' es\_es']t : showStreet!(path, places)\rangle$

$v_{11} = \langle c : getMap?(path, places),$
$[c : lang ==' en\_gb']s : showSatellite!(path, places)\rangle$

$v_{12} = \langle c : getMap?(path, places),$
$[c : lang ==' en\_us']s : showSatellite!(path, places)\rangle$

$v_{13} = \langle c : getMap?(path, places),$
$[c : lang! =' en\_gb' \&\& c : lang! =' en\_us']$
$s : showSatellite!(path, places)\rangle$

$v_{14} = \langle c : switchView!, m : changeView?\rangle$

To express the correspondences between one message on one side (*i.e., [c:loc==null]c:searchMap!(orig,dest)*) and two messages on the other (*i.e., [c:loc==null]m:getOrig?(orig).m:getDest?(dest)*), we use two tuples (v_2 and v_3). The first one is a partial synchronisation with the emission and the first parameter received, and the second one completes the synchronisation by receiving the second parameter expected. The context attribute $\#(c : bw\_lang)$ of v_6 is taken for the Map Service from the Client and passed to the Street Service. It refers to the client language, and specifically to the browser language of the client, which is the used one by the Street Service. Therefore, we rename that context attribute to allow that the Street Service use it $(c : bw\_lang\hat{}lang)$.

4.2 Adaptor Protocol Generation

By using the contextual mapping presented above and the CA-STS services, we generate a new CA-STS, called *CA-STS adaptor protocol*, by applying the algorithm described in [18] and considering conditions and contexts.

Adaptor protocol for our scenario. Figure 3 depicts the CA-STS adaptor protocol for our scenario. The initial state is identified by 0 and the final one by 13. Note message directions are reversed because all messages will go through the adaptor, and this latter has to synchronize with these messages using complementary directions.

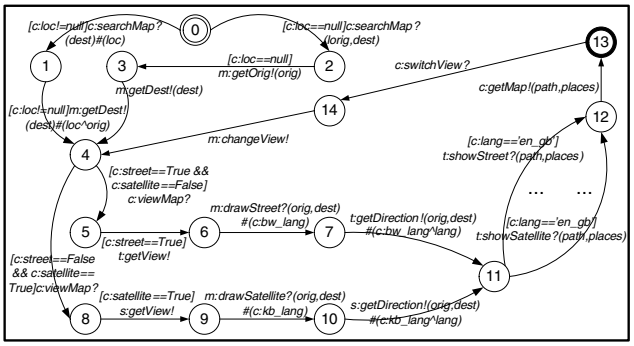

Figure 3. CA-STS adaptor protocol for the running example

Finally, since CA-STS adaptor protocol is a CA-STS service, we apply the verification algorithms for it, to validate its correct execution, and check that it is free of incompatibilities as we expected. Although currently the generation of the CA-STS adaptor protocol has to be performed manually by the developer to take conditions and context information into account, one of the perspectives is that to extend the automatic generation of contracts [17], as well as the algorithm to generate the adaptor protocol [18] (both included in ITACA tool [4]) to support these new features.

5 Related Work

Recently, there have been many research works on context-aware computing, service composition and formal verification of services. However, to the best of our knowledge, only some of them combine their efforts to tackle the three paradigms together. In this work, we focus on all three, not only describing a context-aware service composition model, but also subjecting it to automated verification to detect inconsistencies and potential faults. Different works propose model checking techniques in the Web Service composition field [11, 12, 8, 24]. Some of them even use semantic ontology languages to compose the services. However, none of them tackles context-aware composition.

With respect to service composition, Luo *et al.* [9] present a model to compose services and validate theirs correctness using Petri nets. They check behavioural properties, such as safety, reachability, deadlock and redundancy based on simulation of the model. In contrast, we model the contextual service composition with a strategy by means of CA-STS that may be easily validated. Vukovic presents an approach that focuses on the recomposition of the composite service during its execution, according to changes in the context [23]. It provides a failure-tolerant solution, but user preferences and control of independent requests are not controlled, which our model supports.

On the other hand, some works focus on context-awareness. Firstly, Simcock *et al.* have designed and implemented a mobile and context-sensitive tourist guide system [10]. However, they focus on design and usability issues, not on the services composition. Mokhtar *et al.* [19] present an approach to the context-aware dynamic composition of services to perform user tasks. But no testing mechanism is given to detect inconsistencies, such as we do in our approach. Kim and Choi [16] suggest a context infrastructure to provide semantic interoperability in ubiquitous computing. They evaluate its performance without considering possible un-modeled situations. A model-driven approach to model a contextual and variable process of an application structure, as well as its behaviour and its architecture, is defined by Ayed *et al.* [2]. They have evaluated their approach only by implementing the UML profile, but not using verification techniques. Some work similar to our own has been done in verifying requirements engineering. Heitmeyer *et al.* use finite-state models to discover inconsistencies in SCR specifications [14]. While the classes of inconsistencies they that detect are characteristic of requirements spec-

ifications, the fault patterns that we detect are characteristic of service composition with contextual info.

6 Conclusion and Future Work

In this paper we have presented the formalism of CA-STS to model context-aware Web Services. This model does not only solve some cases of behavioural mismatches, but also helps in distinguishing between available contexts when translating the messages among services, by avoiding faults or inconsistence situations. 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 approach provides to achieve message translation depending on contexts.

We have introduced a set of algorithms to verify efficiently the CA-STS services against a number of properties patterns (non-determinism, liveness of states and rules, absence of blocking states). Also, we have provided a prototype implementation for these algorithms and we have applied it to the verification of an example, first validating the single CA-STSs and last the CA-STS adaptor service, which solves mismatches, obtained as composition of those services. Our prototype implementation is already available, but we plan to develop our approach into a more mature product. We are also planning to extend existing models for Web Services automatically. The idea is to query existing services with and without contextual values, and to map the Web Services that are context-aware (*e.g.,* by observing the response of each method we can flag the context that has been used). In this way we could automatically extend existing models (even if not perfectly).

In addition, we plan to extend the CA-STS model to fix conditions and context variables with their corresponding services in the interaction without the need of use service identifiers. To do that we need to use a *context stack* in the contextual mapping, and the introduction of constraints on this stack to maintain the correspondences. Another perspective is that to extend the ITACA tool [4] to take conditions and context information into account.

Acknowledgements. This work was supported in part by the projects TIN2008-05932 funded by the Spanish Ministry of Science and Innovation, and P06-TIC-02250 funded by the Andalusian Government, as well as the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/D077273/1 (project UbiVal), EP/E006191/1 and EP/F013442/1, and by the European Commission under the project PLASTIC. David Rosenblum holds a Wolfson Research Merit Award from the Royal Society. We gratefully acknowledge the insights provided for this work by Sebastian Elbaum and Sonia Ben Mokhtar.

References

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

[2] D. Ayed, D. Delanote, and Y. Berbers. MDD Approach for the Development of Context-Aware Applications. In *Proc. of CONTEXT'07*, volume 4635 of *LNAI*, pages 15–28, 2007.

[3] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[4] J. Cámara, J.A. Martín, G. Salaün, J. Cubo, M. Ouderni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services (Formal Demo Paper). In *Proc. of ICSE'09*, pages 627–630. IEEE Computer Society, 2009.

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

[6] J. Cubo, C. Canal, and E. Pimentel. Towards a Model-Based Approach for Context-Aware Composition and Adaptation: A Case Study using WF/.NET. In *Proc. of MOMPES'08*, pages 3–13. IEEE Computer Society, 2008.

[7] F. Curbera et al. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003.

[8] J. Garcia-Fanjul et al. Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. In *Proc. of TAIC-PART'08*, pages 127–130. IEEE Computer Society, 2006.

[9] N. Luo et al. Towards Context-Aware Composition of Web Services. In *Proc. of GCC'06*, pages 494–499. IEEE Computer Society, 2006.

[10] T. Simcock et al. Developing a location based tourist guide application. In *ACSW Frontiers'03*, pages 177–183. Australian Computer Society, Inc., 2003.

[11] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *Proc. of ESEC/FSE'07*, pages 225–234. ACM Press, 2007.

[12] C. Gao, R. Liu, Y. Song, and H. Chen. A Model Checking Tool Embedded into Services Composition Environment. In *Proc. of GCC'06*, pages 355–362. IEEE Computer Society, 2006.

[13] Google. http://mail.google.com/. Accessed on 9 October 2008.

[14] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[15] M. Keidl and A. Kemper. Towards Context-Aware Adaptable Web Services. In *Proc. of WWW'04*, pages 55–65. ACM Press, 2004.

[16] E. Kim and J. Choi. A Semantic Interoperable Context Infrastructure using Web Services. In *Proc. of ICCSA'07*, volume 4706 of *LNCS*, pages 839–848, 2007.

[17] A. Martín and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, ENTCS. Elsevier, 2008.

[18] R. Mateescu, P. Poizat, and Gwen 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.

[19] S. Ben 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, 2006.

[20] M. Sama, D. Rosenblum, Z. Wang, and S. Elbaum. Model-Based Fault Detection in Context-Aware Adaptive Applications. In *Proc. of ESEC/FSE'08*. ACM Press, 2008.

[21] K. Scribner. *Microsoft Windows Workflow Foundation: Step by Step*. Microsoft Press, 2007.

[22] F. Somenzi. CUDD: CU Decision Diagram Package - Release 2.4.1. `http://vlsi.colorado.edu/~fabio/CUDD`. Accessed on 9 October 2008.

[23] M. Vukovic. Context Aware Service Composition. Technical Report UCAM-CL-TR-700, University of Cambridge, 2007.

[24] H. Q. Yu and S. Reiff-Marganiec. Semantic Web Services Composition via Planning as Model Checking. Technical Report CS-06-003, University of Leicester, 2006.