HAL
open science

Handling Data-Based Concurrency in Context-Aware Service Protocols

Javier Cubo, Ernesto Pimentel, Gwen Salaün, Carlos Canal

To cite this version:

Javier Cubo, Ernesto Pimentel, Gwen Salaün, Carlos Canal. Handling Data-Based Concurrency in Context-Aware Service Protocols. 9th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'10), Sep 2010, Paris, France. pp.62-77. inria-00539024

HAL Id: inria-00539024

https://inria.hal.science/inria-00539024v1

Submitted on 23 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Handling Data-Based Concurrency in Context-Aware Service Protocols *

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

Ernesto Pimentel
Dept. Computer Science
Univ. of Málaga, Spain
ernesto@lcc.uma.es

Gwen Salaün
INRIA-Grenoble
INP-LIG, France
Gwen.Salaun@inria.fr

Carlos Canal
Dept. Computer Science
Univ. of Málaga, Spain
canal@lcc.uma.es

Dependency analysis is a technique to identify and determine data dependencies between service protocols. Protocols evolving concurrently in the service composition need to impose an order in their execution if there exist data dependencies. In this work, we describe a model to formalise context-aware service protocols. We also present a composition language to handle dynamically the concurrent execution of protocols. This language addresses data dependency issues among several protocols concurrently executed on the same user device, using mechanisms based on data semantic matching. Our approach aims at assisting the user in establishing priorities between these dependencies, avoiding the occurrence of deadlock situations. Nevertheless, this process is error-prone, since it requires human intervention. Therefore, we also propose verification techniques to automatically detect possible inconsistencies specified by the user while building the data dependency set. Our approach is supported by a prototype tool we have implemented.

1 Introduction

Service composition is a crucial paradigm in Service Oriented Computing (SOC), since it allows to build systems as a composition of pre-existing software entities, COTS (*Commercial-Off-The-Shelf applications*) rather than programming applications from scratch. An important issue of service composition is to find out services with capabilities compatible to the user requirements in order to compose them correctly. In a traditional distributed environment, in which all the requests are served in the same way, 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 user and environment information on the requesting client. Thus, context-awareness enables a new class of applications in mobile and pervasive computing, providing relevant information to users. Therefore, context information can help users to find nearby services, to decide the best service to use, to control reaction of systems depending on certain situations, and so on.

Services are accessed through their public interfaces that may distinguish four interoperability levels [9]: (i) the *signature level* provides operation names, type of arguments and return values, (ii) the *behavioural or protocol level* specifies the order in which the service messages are exchanged with its environment, (iii) the *service level* deals with non-functional properties like temporal requirements, resources, security, etc., and (iv) the *semantic level* is concerned about service functional specifications (*i.e.*, what the service actually does). In industrial platforms, service interfaces are usually specified using signatures (*e.g.*, WSDL[1]), but some recent research works [1, 6, 11, 27] have extended interfaces with a behavioural description or protocol. Protocols are essential because erroneous executions or deadlock

*This work is partially supported by the projects TIN2008-05932 and P06-TIC-02250 funded by the Spanish Ministry of Science and Innovation (MICINN) and FEDER, and the Andalusian local Government, respectively.

[1] http://www.w3.org/TR/wsdl

M.R. Mousavi and G. Salaün (Ed.): International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2010).
EPTCS 30, 2010, pp. 62–77, doi:10.4204/EPTCS.30.5

© J. Cubo, E. Pimentel, G. Salaün & C. Canal
This work is licensed under the
Creative Commons Attribution License.

situations may occur if the designer does not take them into account while composing clients and services [18, 24]. In this way, service protocols evolving concurrently in a composition need to impose an order in their execution if there exist data dependencies. Dependency analysis is a technique to identify and determine data dependencies between service protocols. To the best of our knowledge, not many works have tackled the handling of concurrent interactions of service protocols through dependency analysis [5, 10, 14, 17, 26].

In this work, we focus on systems that consist of clients[2] (users with a mobile device such as a PDA or a smart phone) and services modelled with interfaces constituted by context information, a signature, and a protocol description (taking conditions into account). We also consider a semantic representation of service instead of only a syntactic one. We use OWL-S ontologies[3] to capture the semantic description of services by means of relationships between concepts within a specific domain. In order to address the concurrency in the service composition in these systems, we first formalise a model for context-aware clients and service protocols. Second, we propose an approach to handle dynamically the concurrent execution of context-aware service protocols on the same user device, using mechanisms based on data semantic matching. Our approach aims at assisting the user in establishing priorities between these dependencies, avoiding the occurrence of deadlock situations. Constraints on the concurrent execution can be specified using a composition language which defines operators for executing a sequence of protocols, a non-deterministic choice between protocols, and for controlling the data dependencies existing among several protocols executed at the client level at the same time. In addition, since this process requires human intervention (error-prone), we use analysis techniques to automatically verify the correct execution order of the protocols with respect to the built data dependency sets. Our approach is supported by a prototype tool we have implemented. To evaluate the benefits of our approach, we have applied it to different case studies. We analyse the experimental results obtained either with manual or interactive specification of data dependencies and their corresponding execution priorities.

The rest of this paper is structured as follows. Section 2 presents our model formalising context-aware clients and service protocols. In Section 3, we introduce a case study we use throughout the paper for illustration purposes. Section 4 presents the handling of concurrent interactions of context-aware service protocols. Section 5 describes the `ConTexTive` prototype tool that implements our approach, and shows some experimental results. Section 6 compares our approach to related works. Finally, Section 7 ends the paper with some concluding remarks.

2 Context-Aware Service Model

2.1 Interface Model

Our model describes client and service interfaces using context profiles, signatures and protocols. Context profiles define information which may change according to client preferences and service environment. Signatures correspond to operations profiles. Protocols are represented using transition systems.

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"* [13]. Context information can be represented in different ways and can be classified in four main categories [16]: (i) user context: role, preferences, language, calendar, social situation or privileges, (ii) device/computing context: network connectivity,

[2] In the sequel, we use client as general term covering both client and user with a mobile device.

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

device capabilities or server load, (iii) time context: current time, day, month or year, and (iv) physical context: location, weather or temperature. For our purpose, we only need a simple representation where contexts are defined by context attributes with associated values. In addition, we differentiate between static context attributes (*e.g.*, role, preferences, day, ...) and dynamic ones (*e.g.*, network connectivity, current time, location, privileges, ...). Dynamic attributes can change continuously at run-time, so they have to be dynamically evaluated during the service composition. Last, both clients and services are characterised by public (*e.g.*, weather, temperature, ...) and private (*e.g.*, personal data, bandwidth, ...) context attributes. Thus, we represent the service context information by using a *context profile*, which is a set of tuples (CA, CV, CK, CT), where: CA is a context attribute or simply context with its corresponding value CV, CK determines if CA is static or dynamic, and CT indicates if CA is public or private (*e.g.*, (*priv, Guest, dynamic, public*), where *priv* is a public and dynamic context which corresponds to user privileges with *Guest* as value).

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

A *protocol* is represented using a 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 regular 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 either an internal action τ (tau) or 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 which corresponds to the set of CA-STS labels associated to transitions, S is a set of states, $I \in S$ is the initial state, $Fc \subseteq S$ are correct final states (deadlock final states are not considered), and $T \subseteq S \times A \times S$ is the transition function whose elements $(s_1, a, s_2) \in T$ are usually denoted by $s_1 \xrightarrow{a} s_2$.*

Finally, a *CA-STS interface* is constituted by a tuple (CP, SI, P), where: CP is a context profile, and SI is the signature corresponding to a CA-STS protocol P. Both clients and services consist of a set of interfaces. We assume they have several protocols with their corresponding signatures, and a context profile for each one. For instance, let us consider a client with two different protocols P_{c_1} and P_{c_2}. This client consists of two interfaces such as: $I_{c_1} = (CP_{c_1}, SI_{c_1}, P_{c_1})$ and $I_{c_2} = (CP_{c_2}, SI_{c_2}, P_{c_2})$.

We adopt a synchronous and binary communication model (see Section 2.2 for more details). Clients can execute several protocols simultaneously (concurrent interactions). Client and service protocols can be instantiated several times. At the user level, client and service interfaces can be specified using:

(i) context information into XML files for context profiles, (ii) WSDL for signatures, and (iii) business processes defined in industrial platforms, such as Abstract BPEL (ABPEL) [2] or WF workflows (AWF) [12], for protocols. Here, we assume context information is inferred by means of the client requests (HTTP header of SOAP messages), and we consider processes (clients and services) implemented as business processes which provide the WSDL and protocol descriptions.

2.2 Operational Semantics of CA-STS

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

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

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

Function ev_c is defined in a similar way to ev, but it only considers context variables. This is because we first apply ev in order to evaluate all the regular variables:

$$ev_c(E,x) \triangleq E(x)$$

where x is a context variable. We also define an environment overloading operation "$\oslash$" such that, given an environment E, $E \oslash \langle x, v\rangle$ denotes a new environment, where the value corresponding to x is v.

We present in Figure 1 the semantics of one CA-STS ($\rightarrow_o$), with three rules that formalise the meaning of each kind of CA-STS labels: internal actions τ (INT), emissions (EM), and receptions (REC); and one rule to consider the dynamic update of the environment according to the context changes at run-time (DYN). Note that *w.r.t.* Definition 1, $b \in B$ is a condition, $a \in M$ is an operation name, and $x \in F$ and $v \in F$ correspond to a list of variables and expressions, respectively. Condition b may contain regular and/or context variables and both of them must be evaluated in the environment of the source service (sender), because the decision is taken in the sender. However, evaluation of expressions v only affects regular variables (rule EM), since context variables will be evaluated in the target service (receiver) to consider the context values when the message is received (see rule COM in Figure 2). We assume that the dynamic modification of the environment will be determined by different external elements depending on the type of context (*e.g.*, user intervention, location update by means of a GPS, time or temperature update, and so on). Then, we model this situation by assuming a transition relation which indicates the environment update, denoted by $E \rightsquigarrow_d E'$, where $E'(x) \neq E(x)$ only if x is a dynamic context variable.

The operational semantics of n ($n > 1$) CA-STSs ($\rightarrow_c$) is formalised using two rules. A first synchronous communication rule (COM, Figure 2) in which value-passing and variable substitutions rely on a late binding semantics [21] and where the environment E is updated. A second independent evolution rule (INE_τ, Figure 2). A list of pairs $\langle s_i, E_i\rangle$ is represented by $[as_1,\ldots,as_n]$. Rule COM uses the function ev_c to evaluate dynamically in the receiver the context changes related to the dynamic context attributes

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

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

Figure 1: Operational Semantics of one CA-STS

of the sender. Regular variables have been evaluated previously in the rule EM when the message is emitted. This dynamic evaluation handled in the operational semantics allows to model service protocols depending on context changes. Rule INE_τ is executed in case of an internal service propagation that gives rise to either a state (related to the rule INT) or an environment (rule DYN) change. Thus, transitions $\rightarrow_c$ do not distinguish between internal evolutions coming from either internal actions in services or dynamic updates in the environment.

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

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

Figure 2: Operational Semantics of n CA-STSs

3 Motivating Example

For illustration purposes, we consider a road info system that consists of users travelling by car on a road and using mobile devices (called Clients), and Info Services providing information requested by the Clients. Info Services contain information about routes, hotels, restaurants, gas stations, multimedia entertainment such as movies, music, images, shows, and so on, or museums. Some of these services are free (*e.g.*, Route or Gas Station Services) and others have to be payed (*e.g.*, Entertainment or Museum Services). For these latter ones the Client needs to check his/her bank account.

For the sake of comprehension, we consider a reduced part of our case study. Let us suppose that a Client, before starting the trip, wants to plan a route. Afterwards he/she wants to perform at the same time the purchase of both a music album to listen during the trip, and a ticket for a museum located at his/her destination to visit that same day. Ideally, the first request must be satisfied by the nearest Route Service, which considers the context information related to the Client location *loc* (dynamic context attribute), and to the *traffic* and *weather* of the environment (dynamic attributes). The nearest Entertainment Service should manage the second request, by taking into account privileges *priv* (dynamic attribute) of the Client (*e.g.*, if the Client has privileges of subscriber he/she will pay a reduced amount for an album), and its *server load* (dynamic attribute). The third request has to be replied by the Museum Service that also takes into account Client privileges *priv*, and the *day* to visit the museum (static attribute). This last request could also be replied by the Entertainment Service, since this service can handle the purchase of any show (museum, concert, cinema, etc) as well. We consider all the context attributes mentioned are public and in our scenario they have a default value, that in case of the dynamic ones may change.

This scenario requires to discover automatically the most appropriate services for each client's request among the available services (running at the moment of each request) from the repository of Road Info Services. According to the dynamic nature of the context information, context changes at run-time may occur. Thus, for instance, once the Client has requested a route, when some changes in his/her dynamic context attributes (*e.g.*, *loc*) occur, the Route Service situated along the Client's way must automatically recompute the route according to the new context values (rule DYN, Figure 1). On the other hand, we focus on the concurrent executions of several protocols at run-time, that must be handled (*e.g.*, the Client requesting concurrently a music album and a museum ticket). All these considerations make our approach appropriate to model this kind of systems and to handle the concurrent interactions of protocols. For the sake of simplicity, we suppose the available services from the repository are Route, Entertainment and Museum Services. In Figure 3, the interfaces of Client and Info Services are given for the scenario previously described.

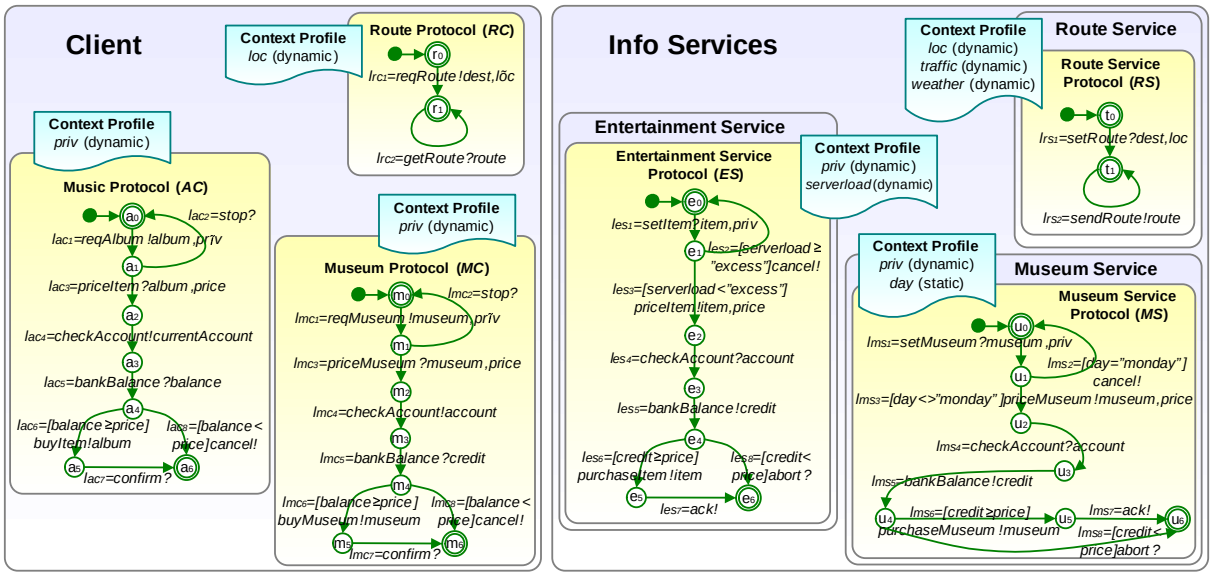

Figure 3: CA-STS Protocols of Client and Info Services for our Scenario

The Client has three interfaces corresponding to the three client's requests (Route, Music (Album) and Museum), which consist of three protocols (*RC*, *AC* and *MC*, respectively), each one with a context profile, and a signature. The latter one will be left implicit, yet it can be inferred from the typing of arguments (made explicit here) in CA-STS labels. On the other hand, each service (Route, Entertainment and Museum) has an interface with a context profile, a (implicit) signature and a protocol (*RS*, *ES* and *MS*, respectively). We assume *RC* should interact with *RS*, *AC* with *ES*, and *MC* with *MS*. It is worth mentioning that the *ES* protocol may be instantiated for communicating with different client's requests related to movies, music, images, shows and so on. Thus, *ES* could also manage the Client's Museum request *MC*. Let us consider, *e.g.*, the label $RC : l_{rc_1} = reqRoute!dest, \tilde{loc}$ from the Client's Route protocol, where *dest* is a data term which indicates the destination requested for the route, and $\tilde{loc}$ is a dynamic context attribute of the Client's Route context profile. The Route Service protocol *RS* receives the request through a label such as $RS : l_{rs_1} = setRoute?dest, loc$ where *dest* and *loc* are variables.

Figure 4 gives the domain ontology related to this road info 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. This ontology has been generated using Protégé 4.0.2[4].

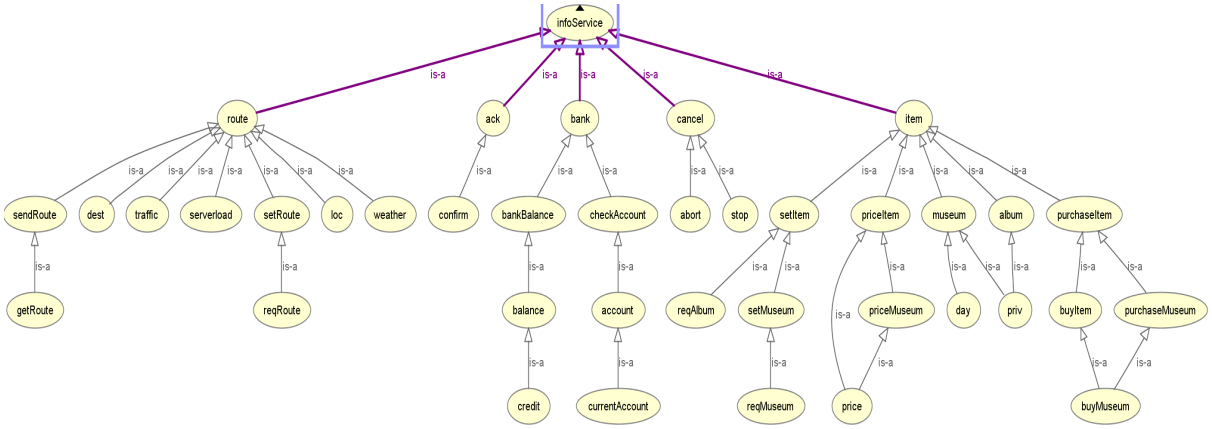

Figure 4: Road Info System Ontology generated using Protégé 4.0.2

4 Handling Concurrent Interactions

This section describes a composition language that allows to execute and handle concurrently interactions between a client and several services at the same time. Our language addresses data dependency issues that appear in the concurrent execution of client protocols, since all data received by a client are shared and can be accessed by several of his/her protocols. Therefore, our mechanism allows to maintain data consistency, even if a change occurs at run-time. We can also detect problems coming from the data dependencies, which would result in deadlocks during the execution of the protocols if not corrected.

4.1 Composition Language

In this section, we formalise a language to dynamically compose several protocols, with the following operators: *sequence*, *choice* and *parallel dependency* (or concurrency).

4.1.1 Syntax

A client can execute a *sequence* of the form $P_1.P_2$, where P_1 and P_2 are two protocols: "execute P_1 and then P_2". A *non-deterministic choice* P_1+P_2 can be performed: "run P_1 or P_2". The concurrent execution of two protocols P_1,P_2 is written $P_1||_{LD}P_2$: "execute P_1,P_2 in parallel while respecting data dependencies specified in *LD*". *LD* is a set of *label dependencies* $\{(id : l > id' : l')\}$, where l an l' are labels, and *id* and id' are protocol identifiers prefixing the labels. *LD* represents dependencies between arguments involved in the labels of these two protocols. Symbol "$>$" indicates the order of execution in which labels must be executed (*e.g.*, $(p_1 : l > p_2 : l')$, l is executed before l'), being l and l' the dominant and dominated labels, respectively. If more than two protocols are executed concurrently, then we will detect the data dependencies by pairs of protocols. Here is the syntax of the composition language:

The goal of our composition language is to illustrate with a minimal expressiveness our service composition approach. We could have also included for instance repetition operators such as $P*$ (executes P

[4]http://protege.stanford.edu/

$$
\begin{array}{lcll}
P & ::= & P_1.P_2 & \textit{sequence} \\
 & | & P_1+P_2 & \textit{non-deterministic choice} \\
 & | & P_1||_{LD}P_2 & \textit{parallel dependency}
\end{array}
$$

several times) or P^x (executes P x times). Nevertheless, repetition can be achieved by launching manually several times the execution of P.

4.1.2 Operational Semantics

We formalise the operational semantics of the composition language. The rules presented in Figure 5 extend the operational semantics of our model to the operators previously considered. In SEQ2, Fc_1 refers to the correct final states of the protocol P_1. Both $+$ and $||_{LD}$ are commutative, therefore the symmetrical rules are omitted. Label l represents either the internal action τ, an emission $a!v$, or a reception $a?x$. PLD1 performs the concurrent execution of the protocols P_1 and P_2 *w.r.t.* a label dependency $(p_1 : l > p_2 : l')$, and removes the label dependencies which include l as first element from the label dependency set LD. PLD2 works as PLD1, but without removing label dependencies, since l appears in a loop in its protocol. Last, PLD3 executes a label which does not belong to the label dependency set.

$$
\frac{\langle s_1,E_1\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle}{\langle s_1,E_1\rangle.\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle.\langle s_2,E_2\rangle}\ \text{(SEQ1)}
\qquad
\frac{\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_2',E_2\rangle \quad s_1 \in Fc_1}{\langle s_1,E_1\rangle.\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_2',E_2\rangle}\ \text{(SEQ2)}
\qquad
\frac{\langle s_1,E_1\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle}{\langle s_1,E_1\rangle + \langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle}\ \text{(NDCH)}
$$

$$
\frac{\begin{array}{c}\langle s_1,E_1\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle \quad (p_1 : l > p_2 : l') \in LD \\ LD' = remove(l,LD) \quad \langle s_1,E_1\rangle \not\xrightarrow{l}_o *\langle s_1,E_1\rangle\end{array}}{\langle s_1,E_1\rangle||_{LD}\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle||_{LD'}\langle s_2,E_2\rangle}\ \text{(PLD1)}
\qquad
\frac{\begin{array}{c}\langle s_1,E_1\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle \quad (p_1 : l > p_2 : l') \in LD \\ \langle s_1,E_1\rangle \xrightarrow{l}_o *\langle s_1,E_1\rangle\end{array}}{\langle s_1,E_1\rangle||_{LD}\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle||_{LD}\langle s_2,E_2\rangle}\ \text{(PLD2)}
$$

$$
\frac{\langle s_1,E_1\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle \quad \forall ld \in LD(p_1 : l \notin get_dominant_label(ld) \wedge p_1 : l \notin get_dominated_label(ld))}{\langle s_1,E_1\rangle||_{LD}\langle s_2,E_2\rangle \xrightarrow{l}_o \langle s_1',E_1\rangle||_{LD}\langle s_2,E_2\rangle}\ \text{(PLD3)}
$$

Figure 5: Operational Semantics of the Composition Language

We define formally the functions used in the operational semantics. Function $remove(l,LD)$ eliminates the label dependencies which include l as first element from the label dependency set $LD = \{ld_1,\ldots,ld_n\}$: $remove(l,\{ld_1,\ldots,ld_n\}) = \{ld_i | ld_{i\in\{1,\ldots,n\}} = (l_1 > l_2) \in \{ld_1,\ldots,ld_n\} \wedge l_1 \neq l\}$

Transition $\langle s_1,E_1\rangle \xrightarrow{l}_o *\langle s_1,E_1\rangle$ is equivalent to $\langle s_1,E_1\rangle \rightarrow_o * \xrightarrow{l}_o \rightarrow_o *\langle s_1,E_1\rangle$, where $\rightarrow_o *$ represents any sequence of transitions $\rightarrow_o$. This will determine if the label l belongs to a loop in transitions in a single protocol, starting from state s and ending in the same state s. Functions *get_dominant_label* and *get_dominated_label* return respectively the dominant and dominated labels from a label dependency:

$get_dominant_label((id : l > id' : l')) = id : l; get_dominated_label((id : l > id' : l')) = id' : l'$

Next, we describe two algorithms to detect label dependencies in concurrent executions of protocols.

4.2 Dependency Analysis

Dependency analysis is a technique to identify and determine data dependencies between service protocols. The main difficulty in analysing dependencies for concurrent executions is how to obtain the relationship between arguments. Protocols evolving concurrently need to impose an order in their execution if there exist data dependencies. A data dependency occurs when a protocol receives a data, which is stored in the user device, and when another client protocol accesses this data (*e.g.*, wants to send it). To detect and handle these dependencies, our semi-automatic dependency analysis process consists of three steps: (i) a first algorithm computes a set of pairs of label dependencies between two protocols, (ii) the user makes a selection among these pairs and determines the order of execution of the selected ones (using the symbol "$>$"), which allows to build an initial label dependency set, and (iii) a second algorithm expands the dependencies chosen by the user to a set as required by the semantic rules PLD1, PLD2 and PLD3 formalised in Figure 5.

The first step is performed by Algorithm 1, that takes as input two protocols, and a domain ontology. It returns all the label dependencies among the argument types of the operation profiles of both protocols. Our algorithm determines that two labels are dependent by using the functions *degree_match*, defined by Paolucci *et al.* [23] (page 339), and *type* to compare their arguments and types, respectively. Function *degree_match* defines four degrees of matching based on semantic matching: {`exact`,`plugIn`,`subsume`, `fail`}. The degree `fail` indicates that the two arguments compared do not match semantically, so we do not consider that there exists a data dependency between them. The remaining three indicate that there is a semantic-based data dependency between the arguments. Function *arguments* in Algorithm 1 returns

Algorithm 1 *pairs_label_dependencies*

returns a set of pairs of label dependencies for two protocols
inputs protocols $P_1 = (A_1, S_1, I_1, Fc_1, T_1)$ and $P_2 = (A_2, S_2, I_2, Fc_2, T_2)$, ontology *Ont*
output a label dependency set LD_p

$LD_p := \emptyset$ // initial value for set of pairs of label dependency
for all $lp_1 \in A_1$ **do**
  $A_{lp_1} := arguments(lp_1)$ // gets the arguments of lp_1
  for all $lp_2 \in A_2$ **do**
    $A_{lp_2} := arguments(lp_2)$ // gets the arguments of lp_2
    $ATD :=$ `false` // by default no dependencies
    for all $arg_{lp_1} \in A_{lp_1}$ **do**
      for all $arg_{lp_2} \in A_{lp_2}$ **do**
        $DM_{arg} := degree_match(arg_{lp_1}, arg_{lp_2}, Ont)$
        $DM_{typ} := type(arg_{lp_1}) = type(arg_{lp_2})$
        if $(DM_{arg} \neq$ `fail`$) \wedge DM_{typ}$ **then**
          $ATD :=$ `true` // argument and type dependency
        end if
      end for
    end for
    if ATD **then**
      $LD_p := LD_p \cup (p_1 : lp_1, p_2 : lp_2)$ // adds a pair
    end if
  end for
end for
return LD_p // returns a set of pairs of label dependencies

all the arguments belonging to a label $l = (b, m, d, f)$: $arguments((b, m, d, f)) = f$

The complexity of Algorithm 1 is quadratic, $O(k \cdot (l \cdot a)^2)$, where k is a constant that indicates the number of dependent labels, and l and a are the average numbers of elements in labels and arguments, respectively. In the second step, the set of pairs of label dependencies returned by the previous algorithm

is showed to the user. The user selects the pairs of label dependencies he/she wants to preserve, and chooses the execution order for each pair. The result is a label dependency set. Given $LD_p = \{(p_1 : l, p_2 : l'), (p_1 : l, p_2 : l'')\}$, if the user: (i) only selects the first pair appearing in LD_p, *i.e.*$(p_1 : l, p_2 : l')$, and (ii) indicates that l' has to be executed before l, then the result will be $LD = \{(p_2 : l' > p_1 : l)\}$.

Last, Algorithm 2 takes as input the two protocols of Algorithm 1 and the set generated in the former step, and returns an extended label dependency set. The algorithm expands the set of label dependencies required by the semantic rules PLD1, PLD2 and PLD3. For each label dependency ld, the algorithm selects all the labels pl_i, $i \in \{1, \ldots, n\}$ preceding the dominant label of ld in the corresponding protocol. Then, for each pl_i the algorithm adds a new label dependency constituted by that pl_i as dominant label and the dominated label of ld as dominated label. For instance, given two protocols P_1 with labels l, l' in sequence and P_2 with l'', if $LD = \{(p_1 : l' > p_2 : l'')\}$ is the label dependency set obtained in the second step, then Algorithm 2 returns a new label dependency set $LD_e = \{(p_1 : l > p_2 : l''), (p_1 : l' > p_2 : l'')\}$.

Algorithm 2 *extended_label_dependencies*

returns an extended set of label dependencies from a label dependency set LD
inputs protocols $P_1 = (A_1, S_1, I_1, Fc_1, T_1)$ and $P_2 = (A_2, S_2, I_2, Fc_2, T_2)$, label dependency set LD
output a label dependency set LD_e

```
LDe := LD // sets the extended set equal to LD
for all ld ∈ LD do
    fl := get_dominant_label(ld) // gets the dominant label
    sl := get_dominated_label(ld) // gets the dominated label
    pf := get_id_protocol(fl) // protocol id of dominant label
    ps := get_id_protocol(sl) // protocol id of dominated label
    PL := get_previous_labels(fl, Tpf) // gets the previous labels to the dominant label fl in the transitions Tpf of its protocol pf
    for all pl ∈ PL do
        if (pf : pl > ps : sl) ∉ LDe then
            LDe := LDe ∪ (pf : pl > ps : sl) // adds a label dependency
        end if
    end for
end for
return LDe // returns the extended set of label dependencies
```

Function *get_id_protocol* gets the protocol identifier of a label $(id : l)$: $get_id_protocol((id : l)) = id$
Function *get_previous_labels* returns the labels preceding a label l in transitions T of a protocol:

$$get_previous_labels(l, T) = \{l' | \exists (s_{i-1}, l_i, s_i) \in T \wedge i = \{1, \ldots, n\} \wedge l_i = l' \wedge l_{n+1} = l\}$$

The complexity of Algorithm 2 is linear, $O(ld \cdot TPL \cdot pl)$, where ld is the number of label dependencies, TPL the average number of transitions to check in the function *get_previous_labels*, and pl the number of labels preceding a concrete label.

Example. Going back to our example, the Client wants to execute the protocol *RC* (route request) in sequence with the parallel execution of the protocols *AC* (music album request) and *MC* (museum ticket request): $RC.(AC||_{LD}MC)$. Our approach builds the set of label dependencies between *AC* and *MC*. First, Algorithm 1 takes as input the two protocols, *AC* and *MC*, and the domain ontology presented in Figure 4. It returns a set of pairs of label dependencies between *AC* and *MC*: $LD_p = \{(l_{ac_4}, l_{mc_4}), (l_{ac_5}, l_{mc_5}\}$ (protocol identifiers in labels have been removed), since for the *checkAccount* operation profile in both *AC* and *MC*, *currentAccount* `exact` matches *account*, and for *bankBalance*, *balance* is semantically compatible to *credit*, with degree of match `plugIn`. However, for instance, for *reqAlbum* and *reqMuseum* of *AC* and *MC* respectively, the degree of match of arguments and types is `fail`, since although *p̃riv* `exact` matches *p̃riv*, *album* `fail` with respect to *museum*. Then, the resulting set is given to the user, who selects the pairs of label dependencies he/she wants to preserve, and chooses the exe-

cution order for each pair. Let us suppose the user only selects the pair (l_{ac_4}, l_{mc_4}) to control the concurrent execution of the operation *checkAccount* in both *AC* and *MC*, by executing l_{ac_4} before l_{mc_4}: $LD = \{(l_{ac_4} > l_{mc_4})\}$. Last, Algorithm 2 takes *LD* as input and extends it with new dependencies needed to execute the semantic rules PLD1, PLD2 and PLD3. Thus, we obtain the final label dependency set: $LD_e = \{(l_{ac_1} > l_{mc_4}), (l_{ac_2} > l_{mc_4}), (l_{ac_3} > l_{mc_4}), (l_{ac_4} > l_{mc_4})\}$. This means that, *e.g.*, for $(l_{ac_1} > l_{mc_4})$, l_{ac_1} is executed before l_{mc_4}, *i.e.*, the label $AC : l_{ac_1} = reqAlbum!album, \tilde{priv}$ is executed before the label $MC : l_{mc_4} = checkAccount!account$, and so on. In such a way, the algorithm controls that l_{mc_4} will not be executed in *MC* until *AC* runs l_{ac_4}.

4.3 Verification of Label Dependencies

The label dependencies construction process is error-prone, since it requires human intervention. This process may provoke possible inconsistencies which result in deadlocks during the execution of the protocols according to the label dependency set computed previously. Therefore, in this section we propose some verification techniques to automatically detect these problems.

To illustrate the need of these verification techniques, we focus on a simple example. In Figure 6, we give, *e.g.*, Client's Planning and Hotel protocols, *PC* and *HC* respectively. The Planning protocol requests for a travel plan to a specific address, and receives a map of the area close to that address. The Hotel protocol searches for a hotel in that map, and gets the destination address. By applying our dependency analysis, Algorithm 1 first returns the pairs of label dependency: $LD_p = \{(l_{ps_1}, l_{hs_2}), (l_{ps_2}, l_{hs_1})\}$. Second, let us suppose the result of the user selection is: $LD = \{(l_{hs_2} > l_{ps_1}), (l_{ps_2} > l_{hs_1})\}$. Last, the extended label dependency set is: $LD_e = \{(l_{hs_1} > l_{ps_1}), (l_{hs_2} > l_{ps_1}), (l_{ps_1} > l_{hs_1}), (l_{ps_2} > l_{hs_1})\}$. In this set, the two

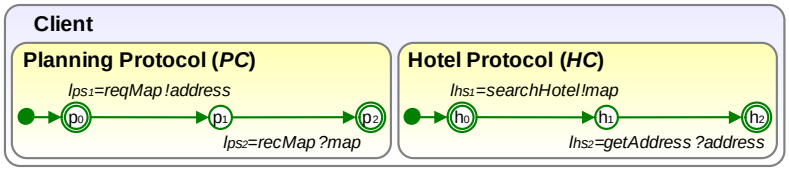

Figure 6: Client's Planning and Hotel Protocols Executing Concurrently

label dependencies $(l_{hs_1} > l_{ps_1})$ and $(l_{ps_1} > l_{hs_1})$ provoke a deadlock, since they are in mutual exclusion. The two (crossed) label dependencies $(l_{hs_2} > l_{ps_1})$ and $(l_{ps_2} > l_{hs_1})$ also generate a deadlock, since the Planning protocol cannot start without the *address* and neither the Hotel protocol without the *map*. The user will be informed to remove one of the label dependencies which provoked this deadlock situation.

Algorithm 3 takes as input two protocols and their label dependency set, and returns a set of traces (pairs of label dependency) leading to deadlock states. To do that, the algorithm compares all the dominant labels from the label dependency set with the dominated ones. If for two label dependencies, a same label is dominant and dominated in both directions or there exist crossed dependencies as described above, then there is a deadlock situation. This problem has to be notified to the user in order to let him/her modify the selection or execution ordering of the label dependencies to avoid that inconsistency.

The complexity of Algorithm 3 is quadratic, $O(ld^2 \cdot TPL)$, where ld indicates the number of label dependencies, and TPL is the average number of transitions to check in the function $get_previous_labels$.

Example. For the scenario of our case study, we applied Algorithm 3 and checked that no problems exist in the label dependency generated in Section 4.2, since there is no trace (pair of label dependency) that provokes a deadlock mismatch when executing concurrently both protocols *AC* and *MC*, *i.e.*, $LD_d = \emptyset$. Therefore, our label dependency set is correct.

Algorithm 3 *label_dependency_verification*

detects possible inconsistencies specified in a label dependency set
inputs protocols $P_1 = (A_1, S_1, I_1, Fc_1, T_1)$ and $P_2 = (A_2, S_2, I_2, Fc_2, T_2)$, label dependency set LD
output a deadlocked label dependency set LD_d

1: $LD_d := \emptyset$ // initial value for set of pairs of deadlocked label dependency
2: **for all** $ldp \in LD$ **do**
3: $fldp := get_dominant_label(ldp)$ // gets the dominant label
4: $sldp := get_dominated_label(ldp)$ // gets the dominated label
5: $p_{fldp} := get_id_protocol(fldp)$ // protocol *id* of dominant label
6: **for all** $ldg \in LD$ **do**
7: **if** $ldp \neq ldg$ **then**
8: $fldg := get_dominant_label(ldg)$
9: $sldg := get_dominated_label(ldg)$
10: $p_{fldg} := get_id_protocol(fldg)$ // protocol *id* of dominant label
11: **end if**
12: **if** $((fldp == sldg) \wedge (sldp == fldg))$
$\vee (sldg \in get_previous_labels(fldp, T_{p_{fldp}}) \wedge sldp \in get_previous_labels(fldg, T_{p_{fldg}}))$ **then**
13: $LD_d := LD_d \cup (ldp, ldg)$ // adds the pair of deadlocked label dependencies
14: **end if**
15: **end for**
16: **end for**
17: **return** LD_d // returns the set of pairs of deadlocked label dependencies

5 Tool Support and Experimental Results

5.1 Tool Support

Our approach for handling concurrency of context-aware service protocols, has been implemented as part of a prototype tool, called `ConTexTive`, which is integrated into our toolbox `ITACA` [8]. `ConTexTive` has been implemented in `Python` with the purpose of being incorporated inside a user device. It aims at discovering services related to a client request and handling the service composition by means of our composition language. We have implemented the algorithms presented in this work, in order to automatically detect data dependencies and check that deadlock situations do not occur when executing protocols concurrently. Figure 7 gives a tool support overview of how our approach has been encoded. Our approach takes client and service interfaces specified as XML CA-STSs, and an XML ontology of a specific domain as input, and detects a label dependency set (*LD*) for each pair of protocols executing concurrently, and checks if this set is consistent (free of deadlocks).

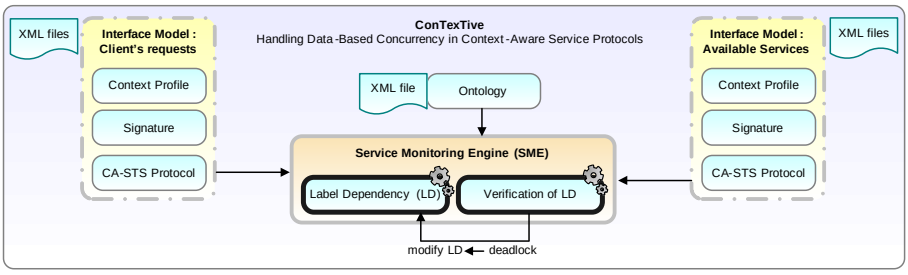

Figure 7: Tool Support Overview

`ConTexTive` has been validated on several examples, such as an on-line computer material store, a travel agency, an on-line booking system or the case study presented here: a road info system.

5.2 Experimental Results

We have conducted a small experimental study with the assistance of a group of volunteers. This study helped us to determine how our approach behaves in terms of evaluating the benefits to find out data dependencies in concurrent executions and to handle those dependencies in terms of effort required, efficiency and accuracy of the dependencies detected. Users performed tests either in a manual or in an interactive (using the tool) way. In order to perform the tests, we provided them a graphical representation of the interfaces and a specific domain ontology to be used in the concurrent interaction, for each problem. Each user solved different problems using different specifications (manual or interactive[5]) to prevent previous user knowledge of a particular case study. Table 1 shows the problems used for our study, which are organised according to increasing size and complexity with respect to the number of interfaces (client and services) involved and the ontology, as well as the overall size of client protocols as a total number of states and transitions. Tests considered all the client protocols interacting concurrently. The table also includes the comparison of experimental results using both manual and interactive specification of data dependencies and their corresponding execution priorities. We consider as parameters the time required to solve the problem (in seconds), the number of label dependencies detected (Depend. in Table 1), and the number of errors in the specified data dependency set.

Problem	Size				Parameter					
	Interfaces		Client Protocols		Time(s)		Depend.		Errors	
	Client	Services	States	Transitions	M	I	M	I	M	I
pc-store-v02	2	2	10	8	61,80	19,14	1	3	2	0
ebooking-v04	2	3	12	13	51,60	3,17	1	1	0	0
roadinfo-v06	3	3	16	18	113,62	16,51	5	4	3	0
travel-agency-v04	3	5	36	36	271,84	62,38	12	12	4	0

Table 1: Experimental Results for the Manual (M) and Interactive (I) Specifications

As it can be observed in the results, there is a remarkable difference in the amount of time required to solve the different problems between manual and interactive specification. We measure as errors the number of wrong, unnecessary or non-detected label dependencies. Our tool always detects all the data dependencies and it uses semantic matching to determine those dependencies, so this is a clear advantage, which increases with the complexity of the problem, compared to the manual specification. Thus, the time elapsed for detecting dependencies by using our tool experiences a linear growth with the size of the problem. Therefore, scalability and performance of our tool are satisfactory, and in the worst case (travel-agency-v04) the time required is roughly 1 minute, which is a reasonable amount of time.

6 Related Work

This section compares our approach to related works by emphasising our main contributions. We successively describe works related to service models by using protocols and/or context information, and works focusing on monitoring of service composition to detect data dependencies in the protocol interaction.

Context-based protocol models address the design and implementation of applications which are able to be modified at the behavioural interoperability level depending on context information. Not many works have been dedicated to model context-aware service protocols. Braione and Picco [7] have

[5]The scenarios were executed on an Intel Pentium CPU 3GHz, 3GB RAM, with Microsoft Windows XP Professional SP2.

proposed a calculus to specify contextual reactive systems separating the description of behaviours and the definition of contexts in which some actions are enabled or inhibited. Related to context-aware adaptation, Autili *et al.* [3] present an approach to context-aware adaptive services. Services are implemented as adaptable components by using the CHAMELEON framework [4]. This approach considers context information at design time, but the context changes at run-time are not evaluated. In our approach, we propose a model to specify protocols based on transition systems and extended with value passing, context information and conditions, which has not been studied yet in previous works. We consider context changes not only at design-time, but also at run-time, since our model allows the continuous evaluation of dynamic context attributes (according to the execution of the operational semantics).

As regards concurrency, models for this discipline emerged, such as CSP [15] and CCS [19], which address concurrent systems from an algebraic perspective. The π-calculus [20] builds on CCS as a process algebra for communicating systems that allows expression of reconfigurable mobile processes. Related to service concurrency, recent approaches have been dedicated to the interaction of services at run-time with the purpose of composing correctly their execution. In addition, several works describe different ways to present data dependencies according to their use for different purposes. Vukovic [25] presents an approach that focuses on the recomposition of the composite service during its execution, according to changes in the context. It provides a failure-tolerant solution, but user preferences and control of independent requests are not controlled, whereas our model supports that. Mrissa *et al.* [22] present a context-based mediation approach to solve semantic heterogeneities between composed Web services by using annotation of WSDL descriptions with contextual details. Their architecture automatically generates and invokes service mediators, so data heterogeneities between services are handled during the composition using semantics and contexts. These works do not handle data dependencies during the concurrent execution of service protocols. Basu *et al.* [5] model such dependencies using a directed edge between nodes. They generate a probabilistic dependency graph as concatenation of all identified dependencies. Ensel [14] presents a methodology to automatically generate service dependency model considering the direction of dependencies. In [17], Kuang *et al.* give a formal service specification describing two types of dependency: dependency on assignment (between the input and output interfaces of an operation), and dependency on sequence (order among operations of a service). Yan *et al.* [26], propose an approach to discover operation dependencies using semantic matching of input and output interfaces and the invoking order among operations. They construct frequency and dependency tables in order to derive indirect dependency relationships by transitive closure algorithm. Most of these approaches do not consider a combination of both the directionality and the execution order to detect dependencies. To the best of our knowledge, the only attempt taking both restrictions into consideration is [26]. Compared to these related works, our approach does not only detect data dependencies, addressing both direction and order, that appeared between communications, but it also allows context-aware protocol concurrent executions at run-time by means of our composition language. In addition, in order to analyse dependencies we rely on semantic matching techniques between data.

7 Concluding Remarks

In this paper, we have described a model to formalise context-aware clients and services. We have also proposed a composition language to handle dynamically the concurrent execution of service protocols. We have defined algorithms to detect data dependencies among several protocols executed on the same user device. These algorithms make possible to establish some priorities on the concurrent execution of protocols affected by these dependencies. In this way, our approach allows to maintain data consistency,

even if a parallel change occurs at run-time. Last, we have proposed verification techniques to automatically detect possible inconsistencies specified by the user while building the data dependency set. We have implemented a prototype tool, `ConTexTive`, which aims at handling the concurrent interaction of service protocols at run-time. Our approach focus on mobile and pervasive systems.

We are currently working on avoiding the human intervention in the process of building the data dependency set by means of priorities previously defined. This will allow to determine automatically the execution order of the detected dependencies, reducing the time required in the interactive specification. We are also extending our approach to solve other problems arisen in the context-aware service composition, such as exception or connection loss. As regards future work, our main goal is to incorporate our prototype tool inside a user device in order to support concurrency in real-world applications running on mobile devices. We also plan to extend our framework to tackle dynamic reconfiguration of services, handling the addition or elimination of both services and context information. In addition, we want to include the repetition operators in our composition language, and to handle concurrent execution of more than two protocols by detecting at the same time all data dependencies existing among several protocols.

References

[1] L. de Alfaro & T. A. Henzinger (2001): *Interface Automata*. In: *Proc. of ESEC/FSE'01*, ACM Press, pp. 109–120.

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

[3] M. Autili, P. Di Benedetto & P. Inverardi (2009): *Context-Aware Adaptive Services: The PLASTIC Approach*. In: *Proc. of FASE'09*, *LNCS* 5503, Springer, pp. 124–139.

[4] M. Autili, P. Di Benedetto, P. Inverardi & F. Mancinelli (2008): *Chameleon Project*. SEA group.

[5] S. Basu, F. Casati & F. Daniel (2007): *Web Service Dependency Discovery Tool for SOA Management*. In: *Proc. of SCC'07*, IEEE Computer Society, pp. 684–685.

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

[7] P. Braione & G.P. Picco (2004): *On Calculi for Context-Aware Coordination*. In: *Proc. of COORDINATION'04*, *LNCS* 2949, Springer, pp. 38–54.

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

[9] C. Canal, J. M. Murillo & P. Poizat (2006): *Software Adaptation: an Introduction*. *L'Objet* 12(1).

[10] J. Cubo, C. Canal, E. Pimentel & G. Salaün (2009): *A Formal Model and Composition Language for Context-Aware Service Protocols*. In: *Proc. of CASTA'09*, ACM Digital Library, pp. 17–20.

[11] J. Cubo, G. Salaün, J. Cámara, C. Canal & E. Pimentel (2007): *Context-Based Adaptation of Component Behavioural Interfaces*. In: *Proc. of COORDINATION'07*, *LNCS* 4467, pp. 305–323.

[12] J. Cubo, G. Salaün, C. Canal, E. Pimentel & P. Poizat (2007): *A Model-Based Approach to the Verification and Adaptation of WF/.NET Components*. In: *Proc. of FACS'07*, *ENTCS* 215, Elsevier, pp. 39–55.

[13] A.K. Dey & G.D. Abowd (2000): *Towards a Better Understanding of Context and Context-Awareness*. In: *Proc. of Workshop on the What, Who, Where, When and How of Context-Awareness*, pp. 304–307.

[14] C. Ensel (2001): *A Scalable Approach to Automated Service Dependency Modeling in Heterogeneous Environments*. In: *Proc. of EDOC'01*, IEEE Computer Society, pp. 128–139.

[15] C. A. R. Hoare (1985): *Communicating Sequential Processes*. Prentice-Hall.

[16] S. Kouadri & B. Hirsbrunner (2003): *Towards a Context-Based Service Composition Framework*. In: *Proc. of ICWS'03*, pp. 42–45.

[17] L. Kuang, J. Wu, Y. Li, S. Deng & Z. Wu (2007): *Exploring Dependency between Interfaces in Service Matchmaking*. In: *Proc. of SCC'07*, IEEE Computer Society, pp. 506–513.

[18] R. Mateescu, P. Poizat & G. Salaün (2008): *Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques*. In: *Proc. of ICSOC'08*, *LNCS* 5364, pp. 84–99.

[19] R. Milner (1980): *Calculus of Communicating Systems*. *LNCS* 2, Springer-Verlag.

[20] R. Milner (1999): *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press.

[21] R. Milner, J. Parrow & D. Walker (1993): *Modal Logics for Mobile Processes*. *Theor. Comput. Sci.* 114(1), pp. 149–171.

[22] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg & S. Dustdar (2007): *A Context-Based Mediation Approach to Compose Semantic Web Services*. *ACM Transactions on Internet Technology* 8(1), pp. 4:1–4:23.

[23] M. Paolucci, T. Kawamura, T.R. Payne & K. Sycara (2002): *Semantic Matching of Web Services Capabilities*. In: *Proc. of ISWC'02*, *LNCS* 2342, pp. 333–347.

[24] F. Plasil & S. Visnovsky (2002): *Behavior Protocols for Software Components*. *IEEE Transactions on Software Engineering* 28(11), pp. 1056–1076.

[25] M. Vukovic (2007): *Context Aware Service Composition*. Technical Report UCAM-CL-TR-700, University of Cambridge.

[26] S. Yan, J. Wang, C. Liu & L. Liu (2008): *An Approach to Discover Dependencies between Service Operations*. *Journal of Software* 3(9), pp. 36–43.

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