A Formal Model and Composition Language for Context-Aware Service Protocols

Javier Cubo, Carlos Canal, Ernesto Pimentel and Gwen Salaün
Dept. of Computer Science, University of Málaga, Spain
{cubo,canal,ernesto,salaun}@lcc.uma.es

ABSTRACT

We first define a model to formalise context-aware clients and services. Then, we propose a composition language available on the user's device to execute and handle concurrently interactions with several services at the same time.

Categories and Subject Descriptors

H.1 [**Models and Principles**]: Miscellaneous; D.2.10 [**Software Engineering**]: Design—*methodologies*

General Terms

Algorithms, Design, Languages

Keywords

Context-Awareness, Service Protocol, Composition Language

1. INTRODUCTION

Context-awareness enables a new class of applications in mobile and pervasive computing, providing the most relevant information to users, and adapting themselves to their situation and preferences. Context information can help users to find nearby services, decide the best service to use (according to the location, connectivity, bandwidth, etc.), control reaction of systems depending on certain situations, find people with similar interests, and so on. Thus, a system using context information can be easily self-adaptive [4], by reducing human effort in the human-computer interaction.

We focus on clients (users with a device) and services modelled with protocols. Protocols are essential because erroneous executions or deadlock situations may occur if the designer does not take them into account while composing clients and services [2, 6]. We also consider context information as well as conditions in protocols that specify how applications should react to context changes. We first formalise a model for context-aware clients and services. Second, we propose a composition language incorporated inside the user's device to handle at run-time the concurrent execution of the client with several services. This language defines an operator controlling the data dependencies existing among different protocols executed at the client level.

CASTA'09, August 24, 2009, Amsterdam, The Netherlands

The rest of this paper is structured as follows: Section 2 presents our model formalising context-aware clients and services. Section 3 describes our composition language. Section 4 presents a case study to illustrate our proposal. Last, Section 5 ends the paper with some concluding remarks.

2. CONTEXT-AWARE SERVICE MODEL

In this section, we formalise the syntax and operational semantics of our context-aware service model.

2.1 Interface Model

In our approach, a system consists of context-aware clients and services. We assume that clients and services are specified using a context profile, signatures and protocols. Context profile defines information depending on client and service environment. Signatures correspond to operation profiles. Protocols are represented by means of Labelled Transition Systems (LTSs) extended with value passing [2], context variables and conditions, that we call Context-Aware Symbolic Transition System (CA-STS). In our model, contexts are called *context attributes*. We represent the service context information by using a context profile, which is constituted of a set of context attributes with associated values. Both clients and services are characterised by public (weather, temperature, season, ...) and private (personal data, bandwidth, local resources, ...) context attributes of their context profile. We also differentiate static context attributes (*e.g.*, role, preferences, day, ...) and dynamic ones (*e.g.*, connectivity, time, location, temperature, ...). Dynamic attributes can change continuously at run-time, therefore they have to be dynamically evaluated during the service composition.

Definition 1 (Context Profile) *A Context Profile is a set of tuples* (CA, CV, CT, CP)*, where:* CA *is a context attribute or simply context (e.g., language) with its corresponding value* CV *(e.g., english),* CT *determines if* CA *is static or dynamic, and* CP *indicates that* CA *is public or private.*

Definition 2 (Signature) *A Signature is 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, with its argument types and its return type.*

Definition 3 (Variable) *A variable is defined as* $Y ::= p|\tilde{u}$*, where:* p *are regular variables, which include static context attributes, and* $\tilde{u}$ *are context variables corresponding to dynamic context attributes.*

Definition 4 (Expression) *An expression is defined as* $X ::= Y|f(X_1, \ldots, X_n)$*, where:* Y *is a variable, and* f *is a function* $f \in \Sigma_f$*.*

Definition 5 (CA-STS Label) *A label corresponding to a transition of a CA-STS is either an internal action (tau) or a tuple* (B, M, D, X) *representing an event, where:* B *is a condition (represented by a boolean expression),* M *is the message name,* D *is the direction of messages (! and ? represent emission and reception respectively), and* X *is a list of expressions if the message corresponds to an emission, or a list of variables if the message is a reception.*

Definition 6 (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, and* $T : S \times A \times S$ *is the transition function whose elements are noted by the expression* $s_1 \xrightarrow{a} s_2$*.*

Definition 7 (Service Interface) *A service interface is a tuple* (CP, S, P)*, where:* CP *is a context profile, and* S *is a signature with its corresponding CA-STS protocol* P*.*

Our communication model is synchronous and binary (see Section 2.2 for more details). Clients can execute several protocols simultaneously (concurrent interactions). Client and service protocols can be instantiated several times. Client and service interfaces can be specified using: (i) XML files for context profiles, (ii) WSDL for signatures, and (iii) business processes defined in industrial platforms, *e.g.*, Abstract BPEL (ABPEL) or WF workflows (AWF) [1], for protocols.

2.2 Operational Semantics of CA-STS

In the following, we use a couple $\langle s, E\rangle$ to represent an active state $s \in S$ and an environment E. An environment is a set of couples $\langle x, v\rangle$ where x is a variable, and v is the corresponding value of x. We use boolean expressions b to describe CA-STS conditions. We also use a function *type* which returns the type of a variable. Two evaluation functions are used to evaluate expressions into an environment: (i) ev evaluates regular variables or boolean expressions, and (ii) ev_c evaluates context variables changing dynamically. We define ev (ev_c is the same as ev for contexts) and the environment overloading "$\oslash$" as follows:

$$E \oslash \langle x, v\rangle \triangleq E(x) = v \qquad ev(E, x) \triangleq E(x)$$
$$ev(E, f(v_1, \ldots, v_n)) \triangleq f(ev(E, v_1), \ldots, ev(E, v_n))$$

We present in Figure 1 the semantics of a CA-STS ($\rightarrow_o$), with three rules that formalise the meaning of each kind of CA-STS labels: τ (TAU), emissions (EM), and receptions (REC). The operational semantics of n ($n > 1$) CA-STSs ($\rightarrow_c$) is formalised using a synchronous communication rule (COM, Figure 2) in which value-passing and variable substitutions rely on a late binding semantics [3], and an independent evolution rule (INE$_\tau$, Figure 2). A list of couples $\langle s_i, E_i\rangle$ is represented by $\{as_1, \ldots, as_n\}$. Rule COM uses the function ev_c to evaluate dynamically in the receiver the context changes of dynamic context attributes of the sender.

3. COMPOSITION LANGUAGE

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 data received by a client are shared and can be accessed by several protocols.

3.1 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 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'). Table 1 formalises the syntax of the composition language.

Table 1: Syntax of the Composition Language

P	$::=$	$P_1.P_2$	*sequence*
	$\mid$	$P_1 + P_2$	*non-deterministic choice*
	$\mid$	$P_1\|\|_{LD}P_2$	*parallel dependency*

3.2 Operational Semantics

The rules presented in Figure 3 give an operational semantics to each operator presented in Table 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.

Function *remove* eliminates the label dependencies which includes l as first element from a label dependency set $LD = \{ld_1, \ldots, ld_n\}$:

$$remove(l, \{ld_1, \ldots, ld_n\}) = \{ld_i | ld_i = (l_1 > l_2) \in \{ld_1, \ldots, ld_n\} \wedge l_1 \neq l\}$$

Function *in_a_loop* returns *true* if label l belongs to a loop in transitions T starting from state s, or *false* otherwise:

$$in\_a\_loop(s, l, T) = \exists seq = [l_1, \ldots, l_n] \wedge l \in \{l_1, \ldots, l_n\} \wedge s \xrightarrow{l_1} s_1 \in T \wedge \ldots \wedge s_{n-1} \xrightarrow{l_n} s \in T$$

Functions *get_dominant_label* and *get_dominated_label* return respectively the dominant and dominated labels from a label dependency $(id : l > id' : l')$:

$$get\_dominant\_label((id : l > id' : l')) = id : l$$
$$get\_dominated\_label((id : l > id' : l')) = id' : l'$$

3.3 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 dependency occurs when a protocol receives a data, which is stored in the user's 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

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

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

Figure 1: Operational Semantics of one CA-STS

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

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

Figure 2: Operational Semantics of n CA-STSs

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

$$\frac{\langle s_1, E_1\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle}{\langle s_1, E_1\rangle + \langle s_2, E_2\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle} \text{ (NDCH)}$$

$$\frac{\langle s_1, E_1\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle \quad (p_1 : l > p_2 : l') \in LD \quad LD' = remove(p_1 : l, LD) \quad \neg in\_a\_loop(s_1, l, T_1)}{\langle s_1, E_1\rangle||_{LD}\langle s_2, E_2\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle||_{LD'}\langle s_2, E_2\rangle} \text{ (PLD1)}$$

$$\frac{\langle s_1, E_1\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle \quad (p_1 : l > p_2 : l') \in LD \quad in\_a\_loop(s_1, l, T_1)}{\langle s_1, E_1\rangle||_{LD}\langle s_2, E_2\rangle \xrightarrow{l}_o \langle s_1', E_1\rangle||_{LD}\langle s_2, E_2\rangle} \text{ (PLD2)}$$

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

Figure 3: Operational Semantics of the Composition Language

analysis process consists of three steps: (i) a first algorithm computes a set of pairs of label dependencies between two protocols, (ii) the user selects some of these pairs and chooses their order of execution, 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 (Figure 3).

The first step is performed by Algorithm 1, that takes as input two protocols, and returns all their label dependencies among the argument types of their operation profiles. Our algorithm determines that two labels are dependent by using the functions *degreeOfmatch*, defined by Paolucci *et al.* [5] (page 339), and *type* to compare their arguments and types, respectively. Function *degreeOfmatch* 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 gets all the arguments from a label l: $arguments(l = (b, m, d, x)) = x$

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) selects the first pair, 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 taken as input to Algorithm 1 and the set generated in the former step, and returns an extended label dependency set. Let $(p_2 : l' > p_1 : l)$ be a label dependency where l' and l are called dominant and dominated label, respectively. 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, if we have two protocols P_1 with labels l, l' in sequence and P_2 with l'', and $LD = \{(p_1 : l' > p_2 : l'')\}$ is the label dependency set obtained in the second step, then Algorithm 2 returns a new label dependency set $LD_e = \{(p_1 : l > p_2 : l''), (p_1 : l' > p_2 : l'')\}$.

Function *get_id_protocol* gets the protocol identifier of a dominant or a dominated label $(id : l)$:

$$get\_id\_protocol((id : l)) = id$$

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 $P_2 = (A_2, S_2, I_2, Fc_2, T_2)$
output a label dependency set LD_p

```
1: LD_p := ∅ // initial value for set of pairs of label dependency
2: for all lp_1 ∈ A_1 do
3:    A_lp1 := arguments(lp_1) // gets the arguments of lp_1
4:    for all lp_2 ∈ A_2 do
5:       A_lp2 := arguments(lp_2) // gets the arguments of lp_2
6:       ATD := false // by default no dependencies
7:       for all arg_lp1 ∈ A_lp1 do
8:          for all arg_lp2 ∈ A_lp2 do
9:             DM_arg := degreeOfmatch(arg_lp1, arg_lp2)
10:            DM_typ := type(arg_lp1) = type(arg_lp2)
11:            if DM_arg ≠ fail ∧ DM_typ then
12:               ATD := true // argument and type dependency
13:            end if
14:         end for
15:      end for
16:      if ATD then
17:         LD_p := LD_p ∪ (p_1 : lp_1, p_2 : lp_2) // adds a pair
18:      end if
19:   end for
20: end for
21: return LD_p // returns a set of pairs of label dependencies
```

Algorithm 2 *extended_label_dependencies*

returns an extended set of label dependencies from a LD
inputs protocols P_1, P_2, label dependency set LD
output a label dependency set LD_e

1: $LD_e := LD$ // sets the extended set equal to LD
2: **for all** $ld \in LD$ **do**
3: $fl := get\_dominant\_label(ld)$ // gets the dominant label
4: $sl := get\_dominated\_label(ld)$ // gets the dominated label
5: $fp := get\_id\_protocol(fl)$ // id of protocol of dominant label
6: $sp := get\_id\_protocol(sl)$ // id of protocol of dominated label
7: $PL := get\_previous\_labels(fl, T_{fp})$ // previous labels in P_f
8: **for all** $pl \in PL$ **do**
9: **if** $(p_f : pl > p_s : sl) \notin LD_e$ **then**
10: $LD_e := LD_e \cup (p_f : pl > p_s : sl)$
11: **end if**
12: **end for**
13: **end for**
14: **return** LD_e // returns the extended set of label dependencies

Function *get_previous_labels* returns all the labels preceding a label l in transitions T of a protocol:

$get\_previous\_labels(l, T) = \{l_i | seq = [l_1, \ldots, l_n, l] \wedge l_i \in \{l_1, \ldots, l_n\} \wedge (s, l_1, s_1) \in T \wedge \ldots \wedge (s_{n-1}, l_n, s_n) \in T \wedge (s_n, l, s_{n+1}) \in T\}$

4. CASE STUDY: ROAD INFO SYSTEM

For illustration purposes, let's consider a system that consists of users travelling by car on a road and using safely mobile devices (called clients), and info services providing information requested by the clients, such as routes, restaurants, gas stations, or multimedia entertainments (movies, music, images, ...). As an example, we suppose that a Client, before starting the trip, wants to obtain a route. Afterwards he/she wants to perform the purchase of a movie for his/her kids and to download a song at the same time. To calculate the route, the Route Service considers the context information related to the Client location *loc* (dynamic context attribute) and to the traffic and weather of the environment (also dynamic attributes), that may change. To download the movie and the song, the Entertainment Service takes into account Client privileges *priv* (dynamic attribute), and its server load (dynamic attribute). In addition, the Client language (static attribute) is also taken into account for the movie. Thus, when changes in dynamic context *loc* occur, the Route Service has to recompute the route according to the new location (function ev_c in rule COM, Figure 2). In Figure 4 are given the Client protocols for the scenario previously described. For space reasons, corresponding service protocols are not depicted. *RC* is the

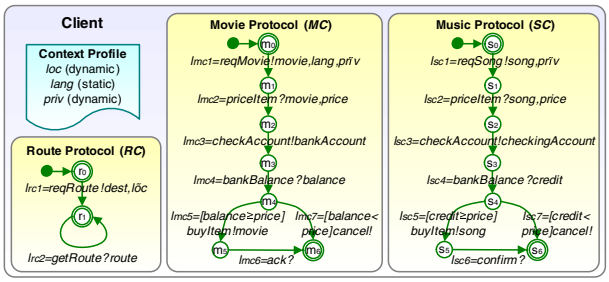

Figure 4: CA-STS Client Protocols for our Scenario

protocol identifier which represents the Route Client protocol communicating with the Route Service. *MC* and *SC* identify the Movie and Music Client protocols respectively, and both interact with the Entertainment Service. We have, *e.g.*, $RC : l_{rc_1} = reqRoute!dest, \tilde{loc}$, 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 context profile. The Route Service protocol *RS* receives the request through an operation profile such as $RS : l_{rs_1} = setRoute?dest, loc$ where: *dest* and *loc* are variables. In our scenario, the Client wants to execute the protocol *RC* in sequence with the parallel execution of the protocols *MC* and *SC*: $RC.(MC||_{LD}SC)$. Now, let's build the set of label dependencies between *MC* and *SC*. First, Algorithm 1 returns a set of pairs of label dependencies between *MC* and *SC*: $LD_p = \{(l_{mc_3}, l_{sc_3}), (l_{mc_4}, l_{sc_4}\}$, since *bankAccount* is semantically similar to *checkingAccount* and *balance* to *credit*, respectively. Then, this set is given to the user, who selects the pairs of label dependencies he/she wants to preserve, and chooses the execution order for each pair. Let's suppose the user only selects the pair (l_{mc_3}, l_{sc_3}) to control the concurrent execution of the operation *checkAccount* in both *MC* and *SC*, by executing l_{mc_3} before l_{sc_3}: $LD = \{(l_{mc_3} > l_{sc_3})\}$. 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: $LD_e = \{(l_{mc_1} > l_{sc_3}), (l_{mc_2} > l_{sc_3}), (l_{mc_3} > l_{sc_3})\}$. This means that, *e.g.*, for $(l_{mc_1} > l_{sc_3})$, l_{mc_1} is executed before l_{sc_3}, *i.e.*, the label $MC : l_{mc_1} = reqMovie!movie, lang, \tilde{priv}$ is executed before the label $SC : l_{sc_3} = checkAccount!checkingAccount$.

5. CONCLUSIONS

In this paper, we have proposed a formal model for context-aware services, and a composition language to handle the concurrent execution of service protocols, by defining algorithms able to detect data dependencies among several protocols executed on a same user's device. Our proposal mainly aims at being applied to pervasive systems and social networks. As regards future work, we first plan to propose verification techniques to automatically detect possible inconsistences specified by the user while building the data dependency set. We also want to handle the execution of a composition specification that can be dynamically modified. Last, we want to detect and solve possible problems raised during the execution, such as exceptions or connection loss.

Acknowledgements. This work has been partially supported by the projects TIN2008-05932 and P06-TIC-02250.

6. REFERENCES

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

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

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

[4] O. Nierstrasz, M. Denker, and L. Renggli. Model-Centric, Context-Aware Software Adaptation. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 128–145, 2009.

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

[6] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE TSE*, 28(11):1056–1076, 2002.