QoS Contract Preservation through Dynamic Reconfiguration: A Formal Semantics Approach

Gabriel Tamura, Rubby Casallas, Anthony Cleve, Laurence Duchien

QoS Contract Preservation through Dynamic Reconfiguration: A Formal Semantics Approach

Gabriel Tamura[a,b,c,*], Rubby Casallas[b], Anthony Cleve[d], Laurence Duchien[a]

[a] *INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, University of Lille 1, Lille, France*
[b] *University of Los Andes, TICSw Group, Cra. 1 N° 18A-10, Bogotá, Colombia*
[c] *Icesi University, ICT Department, Calle 18 122-135 Pance, Cali, Colombia*
[d] *University of Namur, PReCISE Research Center, Belgium*

Abstract

The increasing pervasiveness of computing services in everyday life, combined with the dynamic nature of their execution contexts, constitutes a major challenge in guaranteeing the expected quality of such services at runtime. Quality of Service (QoS) contracts have been proposed to specify expected quality levels (QoS levels) on different context conditions, with different enforcing mechanisms. In this paper we present a definition for QoS contracts as a high-level policy for governing the behavior of software systems that self-adapt at runtime in response to context changes. To realize this contract definition, we specify its formal semantics and implement it in a software framework able to execute and reconfigure software applications, in order to maintain fulfilled their associated QoS contracts. The contribution of this paper is threefold. First, we extend typed-attributed graph transformation systems and finite-state machines, and use them as denotations to specify the semantics of QoS contracts. Second, this semantics makes it possible to systematically exploit *design patterns* at *runtime* by dynamically deploying them in the managed software application. Third, our semantics guarantees self-adaptive properties such as reliability and robustness in the contract satisfaction. Finally, we evaluate the applicability of our semantics implementation by integrating and executing it in FraSCAti, a multi-scale component-based middleware, in three case studies.

1. Introduction

Over the last years, software services have pervaded all aspects of our everyday life. The subsequent proliferation and massive use of these services through ubiquitous computing devices, individually or in combination with traditional systems, challenges the preservation of the associated Quality of Service (QoS) contracts. Moreover, the highly dynamic requirements that appear for these services when confronted with the dynamic nature of their execution contexts further exacerbate the problem of maintaining their expected quality levels (QoS levels) fulfilled under these varying execution conditions. Additionally, these dynamic capabilities are also expected to be reliable and robust to be acceptable.

QoS contracts constitute a natural and effective means for capturing this kind of context-dependent requirements, such as those on performance, availability and confidentiality [1, 2]. Several specifications [3, 4], languages [5, 6], formal semantics [7, 8] and models [9, 10, 11, 12],

*Corresponding author. ADAM Team. INRIA Lille-Nord Europe. Tel: +33-359577865. Fax +33-359577850.
Email addresses: gabriel.tamura@inria.fr (Gabriel Tamura), rcasalla@uniandes.edu.co (Rubby Casallas), acl@info.fundp.ac.be (Anthony Cleve), laurence.duchien@inria.fr (Laurence Duchien)

among others, have been proposed to specify, model and support the QoS contracts elements, the relationship among QoS provisions and requirements, and enforcement mechanisms. However, despite these many advances, the development of a sound theory to preserve QoS contracts in component-based systems still remains an open problem with at least two important challenges. On the one hand, from the components perspective and following its foundations [13, 14], the strategies for contract fulfillment are based on a *per component* basis, depending only on the attributes of the individual components, and then acting on components selected with *ad hoc* strategies [9, 15, 16, 17]. However, as software services result from the interaction of sets of components in a shared computing infrastructure, their QoS properties depend also on the components joint work and their context of execution (e.g., amount of concurrent users and underlying computing infrastructure load). Hence, a first challenge is to preserve QoS contracts with context-aware strategies that dynamically and reliably consider the behavior of sets of components chosen systematically and consistently, and act upon them as a whole.

On the other hand, the treatment of QoS contracts have traditionally focused on what contracts must specify, that is, on guaranteeing QoS obligations. Nonetheless, given the unpredictable nature of context and that QoS attributes depend on it, a second challenge is to fulfill QoS contracts robustly. Robustness is required to maintain the consistency between the contract states and the states of the software subject to the contract conditions over time, especially if the system state deviates from the desired state (e.g., what to do and in which state to leave the system when facing unspecified context situations affecting it?).

In a previous work presented in [18], we proposed a definition of QoS contracts as a *static* mechanism to specify how to reconfigure component-based software systems. In this approach, the QoS contract semantics had to be hard-coded in the reconfiguration system, restricting its applicability and making it impossible to guarantee properties such as robustness. In this paper we formalize the semantics of our QoS contract definition, and implement it in a software framework called QoS-CARE (QoS ContrAct-preserving Reconfiguration systEm). In our definition, a QoS contract establishes the different QoS levels expected to be fulfilled under different context situations, together with respective guaranteeing reconfiguration rules. Thus, to formalize the semantics of QoS contracts, we extend finite-state machines (FSMs) and typed-attributed graph (TAGs) transformation systems, and use them as mathematical denotations for the QoS contract elements and software structures respectively, in two layers. In the *governing* layer, we interpret a QoS contract as an FSM, with expected QoS-levels as states, and reconfiguration rules as transitions. In the *operating* layer, FSM states are mapped to software structures denoted as TAGs, and transitions to software re-configurations via TAG transformations.

The QoS contract semantics formalization allows QoS-CARE to automatically process a contract as the high-level policy that governs the behavior of a self-adaptive system to maintain fulfilled the contracted QoS levels. Moreover, this automation guarantees 5 of the 10 self-adaptation properties that we defined in [19]. Of these 5 properties, short settling-time is related to the practical feasibility of our formal-based approach, whereas atomicity, termination, robustness to context unpredictability, and consistency, to its reliability and robustness. Guaranteeing the remaining 5 properties (i.e., stability, accuracy, small overshoot, scalability and security) is part of our ongoing research work. It is worth noting that, to the best of our knowledge and based on the survey performed in [19], no other approach guarantees more than 4 of the 10 properties.

Concerning the aforementioned challenges, the novelty of our contribution is as follows. For the first challenge, our extension of TAG transformation systems as denotation for software structures and reconfiguration rules makes it possible to exploit *design patterns* (i.e., sets of components consistently considered as a whole with specific goals) at runtime. Despite these patterns

have been applied until now at design-time to determine quality attributes in software systems [20, 21, 22, 23, 24], we encode them in reconfiguration rules to replace them *dynamically* in the running system to fulfill specific QoS levels. For the second challenge, our FSM extension as denotation for QoS contracts guarantees robustness with respect to context unpredictability. Concerning the methodology, our approach introduces the use of two-layered denotational semantics to the specification of QoS contract semantics. Previously, similar methods have been used, nonetheless, for programming languages [25]. In our case, this semantics bridges the abstract and actual running-software worlds, and generalizes the applicability of QoS-CARE.

This paper is organized as follows. Section 2 introduces a motivating application scenario. Section 3 presents an overview of our semantic specification approach. Sections 4 and 5 present the formal definitions for the two layers of denotations used. Section 6 analyzes the properties of QoS-CARE, and Section 7 shows its applicability. Section 8 compares our approach with similar ones, and Section 9 concludes the paper.

2. Application Scenario

To better understand the requirements for preserving the satisfaction of QoS contracts, we use a simplified version of a reliable video conference system (RVCS). For the user, the video conference services are provided through a software application subject to a QoS contract. This contract, negotiated with the user and provided by a software evolution architect, specifies QoS-level obligations for the confidentiality and availability of the RVCS services under different context conditions. In this example, confidentiality and availability are QoS properties interpreted according to Barbacci *et al*. [26]. That is, the RVCS must ensure (i) the confidentiality on video conference transmissions, and (ii) the continued service of active video conferences, with the expected quality levels under their respective context conditions, as illustrated in Table 1.

Table 1: QoS contractual conditions and corresponding Service-Level Objectives (QoS levels) for:

(a) Confidentiality (based on corporate network access)

Context Condition	Expected QoS Level
CC1: Extranet Connection	ConfidentChannel
CC2: Intranet Connection	ClearChannel
CC3: No Netw. Connection	LocalCache

(b) Availability (based on bandwidth in kbit/s)

Context Condition	Expected QoS Level
CC4: $Bandwidth \leq 12$	CallOnHold
CC5: $12 < Bandwidth \leq 128$	VoiceCall
CC6: $128 < Bandwidth$	Voice&Video

Thus, this contract implies that RVCS software clients are responsible for maintaining the application services to their users with the expected quality levels, in a "smart" way. For instance, in the confidentiality case, whenever a software client connects to the application services from an extranet-serviced area (cf. CC1 in the table), the communication channel is expected to be configured with ciphering/deciphering components between client and server. As this is known as having a "confidential channel" structure, the respective expected QoS level is named "ConfidentChannel". Whenever the user moves into an intranet-serviced area, presumably not requiring securing mechanisms in the middle between client and server, a so called "clear channel" structure is expected to be configured in the communication channel (cf. CC2). In case of no network connection, the RVCS client-server communication is expected to be configured with a local/remote cache structure with automatic synchronization capabilities upon communication resumption (cf. CC3), putting the call on hold (respective QoS levels named accordingly).

It is worth noting that addressing these requirements statically (e.g., with *if-then* clauses on context conditions) would not be satisfactory. First, QoS levels can be renegotiated at runtime, for instance introducing new ones (i.e., new states), thus implying the deployment of additional functionalities (i.e., new software components). Second, as the video conference requires bi-directionality between client and server, this would introduce undesirable synchronization problems between the client's and server's code in their maintainability and also in their execution consistency. For instance, lets consider simple conditions of a QoS contract that depend on context situations such as the current bandwidth of the network interface, or the throughput of service requests. Concerning the maintainability, every of these context conditions should be managed explicitly in both the client and the server code to process them accordingly to the responsibilities on each side. Each new condition to be considered in a contract would require a modification of the source code of both sides. Concerning the execution consistency, the question is that context conditions depend on where they are sensed, at runtime: for the example on network bandwidth, different clients located in different places and connected from different network access points sense different values, and these values are different than the sensed values by the server, for the same context condition. If these conditions are managed in the code of both client and server, they must synchronize and agree on the behavior of the response to changes on these conditions, at runtime.

In the next section we present an overview of our approach, highlighting the challenges that we address in this paper.

3. Layered Denotational Semantics for QoS Contracts: An Overview of QoS-CARE

Informally, the semantics of QoS contracts can be globally approached with a finite-state machine (FSM) with the expected QoS levels as states, and the events on context condition changes as transitions. For instance, the confidentiality contract of our application scenario specified in Table 1a can be represented as in Figure 1.

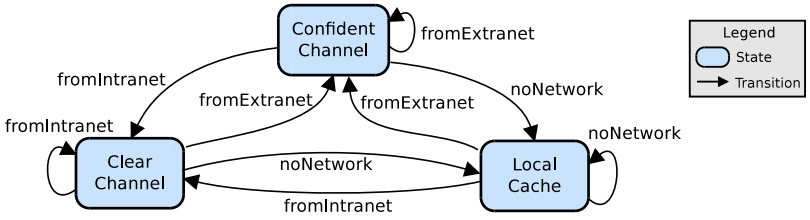

Figure 1: FSM for the confidentiality contract. States represent expected QoS-levels to be fulfilled by the RVCS application under different context situations. On each state, monitor probes in the RVCS software clients notify changes on these situations such as moving into areas with network access *fromIntranet*, *fromExtranet*, or having *noNetwork*.

Nonetheless, even though this tentative approach is plausible and resembles others (e.g., those based on statecharts [27]), it presents two fundamental challenges to achieve the goal of preserving autonomously QoS levels as the corresponding contract semantics. The first challenge is that this approach is "nominal" in the sense that the transitions it performs are confined to the state-machine, that is, they operate only at an abstract level, lacking an explicit and concrete mechanism to achieve the objective of the transitions at the actual software level. Thus, these transitions have no enforcing means for the associated running system to fulfill the expected QoS levels under changing context situations. The second challenge is that this approach does not

guarantee contract robustness with respect to context unpredictability. In this sense, robustness implies to manage both, context events notified by context monitors but unforeseen by the user, as well as the inefficacy or absence of reconfiguration rules in a given contract to fulfill foreseen QoS levels.

We address these challenges with two layers of denotations as described in the next sections.

3.1. Context-Aware Reconfiguration Mechanism

To address the first challenge, our strategy to fulfill QoS levels under different context conditions is to use *design patterns*. These are specific sets of components considered as consistent constructs that determine quality attributes in software systems. For instance, in the confidentiality case of our application scenario we identify three of these sets of components, namely the corresponding to (i) the confidential-channel components structure, (ii) the clear-channel components structure, and (iii) the local/remote-cache components structure. In fact, these structures correspond to design patterns for distributed components and secure communications as presented in [28, 23, 29]. Other design patterns proposed to be used at design-time for addressing quality attributes such as performance and availability also exist in the literature (e.g., [30, 31, 32, 33]).

However, our strategy is to use these design patterns not at design-time, as they have been used until now, but at runtime, by encoding them in the left- and right-hand sides (LHS and RHS respectively) of reconfiguration rules to replace them *dynamically* when the running system transition between states. This replacement is performed through pattern matching of a given rule's LHS on the actual running system, and the dynamic deployment of the modifications implied by the differences between this LHS and its corresponding RHS on the system components and relationships. Nonetheless, instead of performing these operations at the running system level, we use typed-attributed graphs (TAGs) as denotations for the component-based structure (CBS) of the system, and for the LHS and RHS (i.e., design patterns) of reconfiguration rules. Thus, the system reconfiguration operation ($CBS_1 \rightarrow CBS_2$) is abstracted as a graph transformation operation ($TAG_1 \rightarrow TAG_2$) using graph-based pattern-matching. Of course, the benefit of using TAGs as denotations for CBSs is that we can define $f : CBS \rightarrow TAG$ such that, for every pair of vertices (CBS_1, CBS_2) and (TAG_1, TAG_2) with $TAG_1 = f(CBS_1)$ and paths $q = q_1, ..., q_m$ between TAG_1 and TAG_2, from these paths $q_1, ..., q_m$ we can synthesize respective paths $p = p_1, ..., p_n$ between CBS_1 and CBS_2, such that $q \circ f = f \circ p$ and $TAG_2 = f(CBS_2)$. That is, we have found a systematic way to obtain p, the component-based reconfiguration plan, from an abstracted model of the running system (i.e., from the graph transformation q between its formal denotations), an important problem in the software engineering for self-adaptive software systems community [34]. In denotational semantics this is illustrated with the commutative diagram:

$$\begin{array}{ccc} CBS_1 & \xrightarrow{\;p\;} & CBS_2 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ TAG_1 & \xrightarrow[\;q\;]{} & TAG_2 \end{array}$$

In other words, we use design patterns not only as a systematic way to fulfill expected QoS levels under their respective context conditions by modifying the system architecture at the abstract level, but also provide the means to automatically synthesize the reconfiguration plans to instrument these modifications in the actual running system. We call this first denotation the operational layer of our semantic specification.

3.2. Guaranteeing Contract Robustness

To address the second challenge, guaranteeing contract robustness, the strategy has to consider all of the plausible situations that the system can face in its execution concerning the QoS contract definition. To achieve this, we characterize the different types of context conditions that determine not only states and transitions of contract fulfillment, but also of unfulfillment. Therefore, we redefine the usual 5-tuple FSM as a contract-driven tuple FSM, thus called QoSC_{FSM} (QoS Contract-driven FSM), which incorporates both fulfillment and unfulfillment contract super-states and transitions. To guarantee contract robustness, the fulfillment super-state groups all of the expected QoS levels to be fulfilled, which are associated with the specific component-based structure (CBS) configurations (e.g., RVCS with confident channel, clear channel or local/remote cache). Then, the contract denotation is completed with two other super-states that comprise the states of contract unfulfillment, which result from the aforementioned characterization of context conditions. Transitions are associated with TAG reconfiguration rules (i.e., LHS-RHS encoding design patterns) triggered by context events. We call this second layer of denotations the governing layer of our semantic specification.

In other words, we have a two-layered semantics in which the denotations of the second layer are parameterized with denotations of the first one. Thus, our semantic specification of QoS contracts results from a semantic elaboration that combines the semantics of TAGs and FSMs.

3.3. QoS Contract Semantics in Practice

In the application scenario introduced previously, assume initially that a mobile user joins a video conference from her office at the corporate building from an intranet-serviced WiFi access-point. In this initial state, the RVCS application is configured satisfying the respective condition $CC2$ in Table 1a, that is, with a clear channel configuration (cf. (R1) in Figure 2).

A second application state should be reached when the user moves from her office to outside of the company building, accessing the RVCS services from any of the available extranet-serviced access-points, such as GSM or UMTS, thus changing to the context condition $CC1$. This context change, notified by the context event labeled (E) in Figure 2, signals an imminent violation of the confidentiality contract, which is being fulfilled by the current application state. Being in effect the new condition ($CC1$) in the execution context, the respective QoS level (a confident channel) is expected to be fulfilled. Thus, the expected system response is to reconfigure itself to deploy a confidential channel design-pattern, in a transparent way to the user to obtain the (R2) system configuration. However, QoS-CARE performs this reconfiguration by (i) obtaining the TAG denotation (G1) of the actual running-application state; (ii) applying graph pattern-matching and transformation (G*) using the design patterns encoded in the LHS and RHS of reconfiguration rules; (iii) obtaining a new graph representation (G2); and finally, (iv) instrumenting the graph-based reconfiguration operation back into the running application, obtaining (R2). This new configuration, with a confidential channel, should fulfill the expected QoS level $CC1$.

In the following sections we present the details of the TAG and QoSC_{FSM} definitions and elaborate on how we use them to build our two-layered semantics for QoS contracts.

4. Denotation of Component-Based Structures and Design Patterns

The denotations we use for component-based software structures and design patterns correspond to refinements of our formal definitions presented in [18]. In this section we recall them.

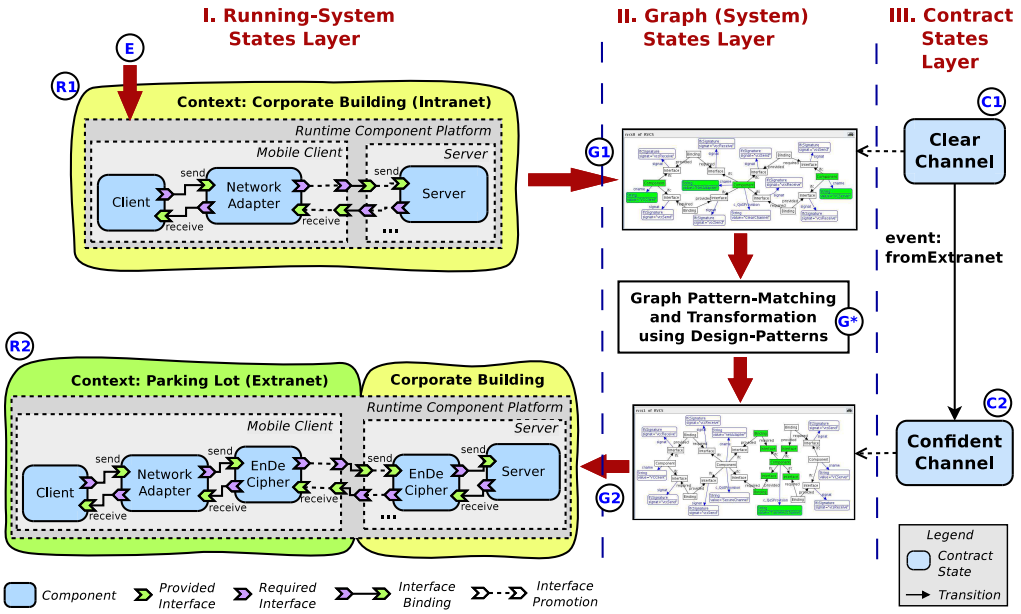

Figure 2: QoS contract semantics in practice. The two layers of denotations for the running-system configurations (e.g., R1 and R2) are shown here vertically as I-TAGs (e.g., G1 and G2); and II-QoSC$_{\text{FSM}}$ (e.g., C1 and C2).

Built on the typed and attributed graph transformation systems by Taentzer [35] and Ehrig [36], our approach defines an abstract syntax for Components-based (CBD[1]) typing structure, system's reflection representation, QoS contracts, and design patterns in reconfiguration rules[2]. Our choice of graphs to denote software structure configurations and its runtime re-configuration has three motivations. First, the expressiveness of graphs to model software structures and design patterns, and to encode them in left and right hand sides of graph transformation rules. Second, the determining relationship between design patterns and specific quality attributes (i.e., QoS levels) [20, 21, 22, 23, 24, 28, 32, 30, 33, 29]. Third, the benefits granted by graph transformation properties.

4.1. System Reflection

The system reflection structure of managed component-based applications is denoted by a set of definitions in terms of typed-attributed graphs (TAGs), which comprise the usual CBD's component, interface, interface type and binding elements. Composites, which allow to scale system complexity by grouping components, are naturally abstracted as components, given that we address structural reconfiguration at the system level. As all of our definitions of software structures are given in terms of TAGs, we start with the TAG definition.

[1]The Components paradigm following Component-Based Development (CBD), Component-Based Software Engineering (CBSE), and Service Component Architecture (SCA).

[2]We use the open source project Attributed Graph-Grammar (AGG) System to represent and process our graph definitions —http://user.cs.tu-berlin.de/~gragra/agg/

Definition 1 (TAG, TAG morphism). A TAG is an tuple $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$, where

- V_1, V_2 are sets of graph and data nodes, respectively;
- E_1, E_2, E_3 are sets of edges (graph, node-attribution and edge-attribution, respectively);
- $source_1 : E_1 \to V_1$; $source_2 : E_2 \to V_1$; $source_3 : E_3 \to E_1$ are the source functions for the edges; and
- $target_1 : E_1 \to V_1$; $target_2 : E_2 \to V_2$; $target_3 : E_3 \to V_2$ are the target functions for the edges.

A TAG morphism t between TAGs G and H, $t : G \to H$, is a tuple $(t_{V_1}, t_{V_2}, t_{E_1}, t_{E_2}, t_{E_3})$ where $t_{V_i} : G_{V_i} \to H_{V_i}$ and $t_{E_j} : G_{E_j} \to H_{E_j}$ for $i = 1, 2$, $j = 1, 2, 3$, such that t commutes with all *source* and *target* functions[3].

As depicted in Figure 3, a TAG adds the usual definition of graph $(V_1, E_1, source_1, target_1)$, with (i) V_2, the set of data-attribution nodes; (ii) E_2 and E_3, the sets of data-attribution edges; and (iii) the corresponding *source* and *target* functions for E_2 and E_3, used to associate the attributes for V_1 and E_1, respectively, to V_2. TAG morphisms can be used to define relationships (such as typing conformity) between TAGs.

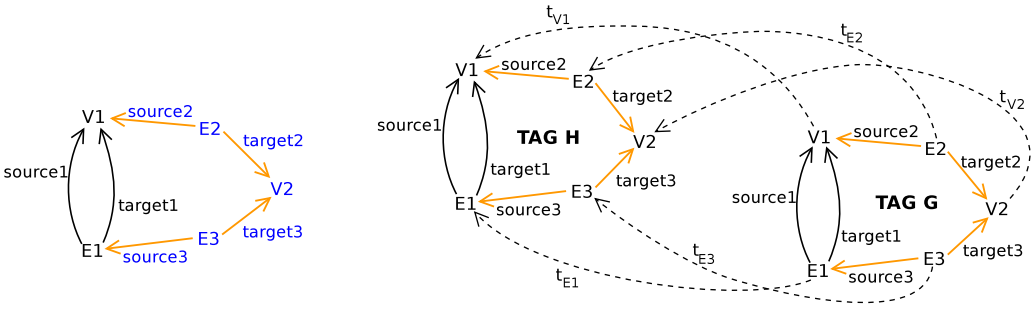

Figure 3: (a) TAG definition and (b) TAG morphism t between TAGs G and H.

Next, to ensure the conformity of TAG-based software structures with respect to the CBD specification, we define the component-based structure type, $\text{CBSTYPE}_{\text{TAG}}$.

Definition 2 ($\text{CBSTYPE}_{\text{TAG}}$). The component-based structure type, $\text{CBSTYPE}_{\text{TAG}}$, is the tuple $(G, DSig)$, where:

- $DSig$ is a data signature over the disjoint union $Integer + String + InterfaceSignature + InterfaceRole$, $InterfaceRole = \{Provided, Required\}$, with the usual CBD interpretations;
- G is the TAG $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that $V_1 = \{Component, Interface, Binding\}$; each of the data nodes in V_2 is named after its corresponding sort in $DSig$, $V_2 = Integer + String + InterfaceSignature + InterfaceRole$; $E_1 = \{ifc, provided, required\}$; $E_2 = \{cname, iname, role, signature, c\_QoSProvision, i\_QoSProvision, ct\_QoSProvision, cmultiplicity, imultiplicity, bmultiplicity\}$; $E_3 = \{ifcmult, pmult, rmult\}$; and functions $(source_i, target_i)_{i=1,2,3}$ as depicted in Figure 4

[3]TAGs combined with TAG morphisms form the category *EGraphs*. See [36] for more details on this.

The component-based structure type is a TAG where graph-nodes represent the usual CBD elements. Graph-edges correspond to the relationships among these elements. Data-nodes represent types of attributes, and data-edges, the typing relationships. *QoSProvision* is a special attribute to annotate components, interfaces and bindings to express their provision of particular QoS capabilities, such as *secure* network connections or *scalable* processing possibilities.

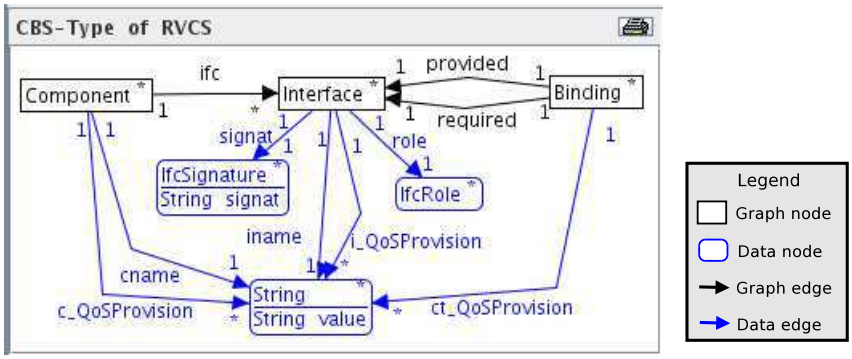

Figure 4: The $\text{CBSTYPE}_{\text{TAG}}$ definition. The *Integer* type and *mult* attributes, for multiplicity constraints, are presented in the usual (short) notation.

Naturally, the purpose of $\text{CBSTYPE}_{\text{TAG}}$ is to serve as a type for the actual CBS instances denoted as TAGs (thus written CBS_{TAG}). CBS_{TAG}, typed with $\text{CBSTYPE}_{\text{TAG}}$, is called the application reflection of the respective managed system because we use it as an internal representation of the system as an indirect means to modify its structure and quality attributes at runtime.

Definition 3 (Component-Based Structure Application Reflection—CBSAR). Given CBS the computational state of the managed component-based software application at runtime, its corresponding reflection state is defined as $\text{CBSAR} = (\text{CBS}_{\text{TAG}}, t)$, where CBS_{TAG} is the TAG that denotes CBS through the one-to-one function $f : \text{CBS} \rightarrow \text{CBS}_{\text{TAG}}$, and t is the TAG morphism $t : \text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$ ensuring that CBS_{TAG} conforms to the typing structure $\text{CBSTYPE}_{\text{TAG}}$.

That is, being CBS the representation of the managed system state in terms of its components, services, interfaces and bindings, as maintained in runtime component platforms, CBSAR encapsulates its corresponding TAG denotation. Recalling the commutative diagram presented in Section 3.1, the purpose of f is to map the component-based system structure to the TAG domain, in which the system reconfiguration is to be operated. The definition of f is crucial for the applicability of our model, given that it bridges the running software world with its corresponding abstract (TAG) world. We use $\text{CBS}_{\text{TAG}}.\mathit{Component}$ as an abbreviation for the set of the graph elements in CBS_{TAG} typed as components by the typing morphism t, that is, $\text{CBS}_{\text{TAG}}.\mathit{Component} = \{c \mid c \in G_{V_1} \wedge t_{V_1}(c) = \mathit{Component}\}$ (analogously for the other elements).

Example 1 (Video Conference Component-Based Structure Application Reflection). Figure 5 illustrates the runtime component-based structure of our video conference example (i.e., its CBS) when connected from an intranet-serviced area (i.e., with a clear-channel connection).

Figure 6 illustrates the corresponding CBS_{TAG} denotation of the system reflection (CBSAR) state, which conforms to $\text{CBSTYPE}_{\text{TAG}}$. The components are represented as exactly the video

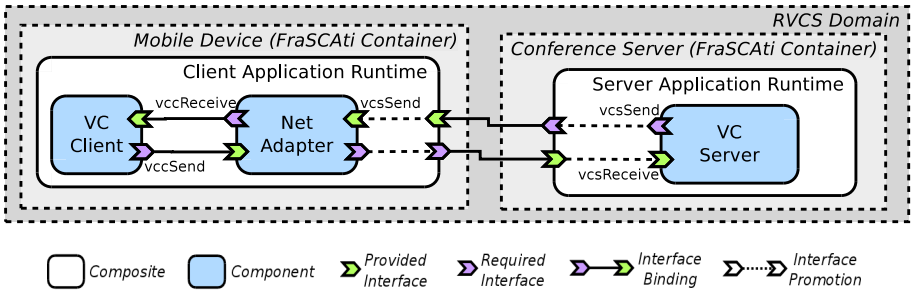

Figure 5: Runtime component-based structure application for Example 1 with a clear channel connection.

conference client (the leftmost highlighted component node with c_name="*VCClient*") with its network connection ("*NetAdapter*") and the server ("*VCServer*"). The other elements represent the interfaces of these components ("*vccSend*", "*vccReceive*" and so on), and their bindings. The component "*NetAdapter*" is provisioned for providing a *"ClearChannel"* network connection, as expressed by its *c_QoSProvision* attribute.

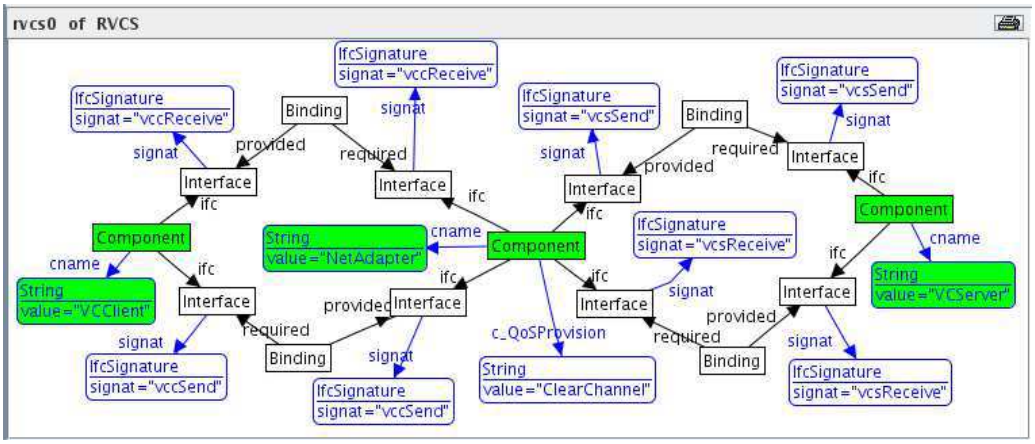

Figure 6: TAG denotation for the component-based structure of the running application depicted in Figure 5.

4.2. QoS Contracts

A QoS contract specifies obligations as expected QoS levels for a given functionality under specific context conditions. These obligations can be seen as the guarantees offered by a system or service provider to *any* of its potential clients ([1, 4, 9, 33, 2]). Thus, a QoS contract is an invariant to be preserved by a system, for instance, by restoring it in case of its violation. The evaluation of the invariant validity must be performed at runtime, given that it depends on measurements from the actual context of execution, such as response time, throughput, and security level on network access location. Therefore, the QoS levels must be monitored and the system must act upon their (imminent) violation in order to have the possibility of maintaining them fulfilled.

Definition 4 (QoS Contract –QoSC). Given $QoSDSig$ the usual data signature over the disjoint union $Integer + String + Boolean + Predicate$, a QoS contract is a tuple (C, ct), where

- C is the TAG that denotes the contract instance, subject to
- ct, the TAG morphism $ct : C \rightarrow \text{QoSCTYPE}_{\text{TAG}}$, where $\text{QoSCTYPE}_{\text{TAG}}$ is the TAG typing definition for QoS contract instances, $\text{QoSCTYPE}_{\text{TAG}} = (V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that $V_1 = \{QoSContract, QoSProperty, QoSMonitor, QoSGuarantor, SLOObligation, QoSRuleSet\}$; each of the data nodes is named after its corresponding sort in $QoSDSig$, $V_2 = Integer + String + Boolean + Predicate$; $E_1 = \{property, obligation, monitor, guarantor, ruleSet\}$; $E_2 = \{gname, pname, mname, rname, SLOPredicate, contextCondition, isActive, contextEvType, QoSCmult, QoSPmult, QoSMmult, QoSGmult, SLOOmult, QoSRmult\}$; $E_3 = \{pmult, omult, mmult, gmult, rmult\}$; and the functions $(source_i, target_i)_{i=1,2,3}$ as depicted in Figure 7.

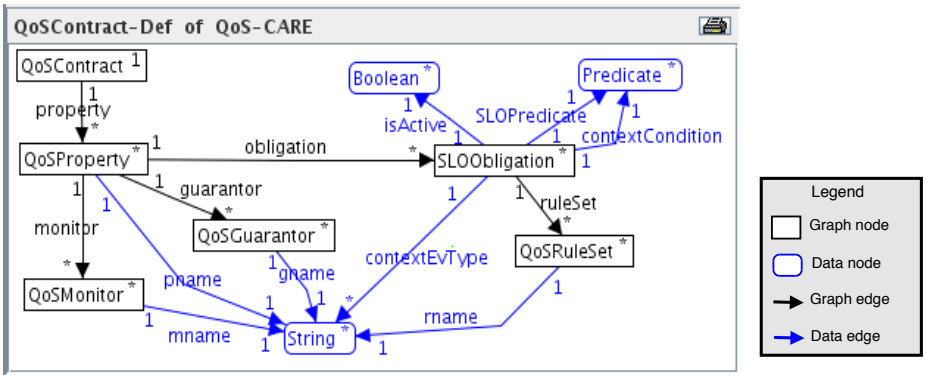

Figure 7: The TAG definition for the QoS Contract Type. The *Integer* type and *mult* attributes, for multiplicity constraints, are presented in the usual (short) notation.

This definition involves the assignment of responsibilities for the coordinated operation of the elements required to perform a software reconfiguration, as established by the autonomic computing vision [37]. That is, a QoS contract is specified referring to a set of *QoSProperties*, each of which has:

- a set of obligations (*SLOObligation*), each with its expected QoS level to be fulfilled (*SLOPredicate*) under a corresponding context condition (*contextCondition*) signaled by context-events of a given type (*contextEvType*); a guaranteeing reconfiguration rule-set (*QoSRuleSet*) to apply upon notification of these events. This rule-set is used to perform the reconfiguration of the component-based structure application reflection state (CBSAR) of the actual running system.
- a set of monitoring elements (*QoSMonitor*) that notify the respective context events; and
- a set of *QoSGuarantors*, system elements distinguished as being the most directly related with the functionalities that determine the fulfillment of the expected QoS levels.

As in the case of CBS instances, QoS contract instances, denoted as TAGs, must have conformance with their typing TAG $\text{QoSCTYPE}_{\text{TAG}}$.

4.3. Component-Based Structure Reconfiguration System

Having defined the denotations for the structural parts of a system in terms of TAGs, we define the runtime software structure reconfiguration as a TAG transformation system. The definition of this reconfiguration system is based on the definition of reconfiguration rule.

Definition 5 (CBS Reconfiguration Rule). A component-based structure reconfiguration rule is defined as $p = (L\text{CBS}_{\text{TAG}}, K\text{CBS}_{\text{TAG}}, R\text{CBS}_{\text{TAG}}, l, r, lt, kt, rt)$, abbreviated $p = (L\text{CBS}_{\text{TAG}} \overset{l}{\leftarrow} K\text{CBS}_{\text{TAG}} \overset{r}{\rightarrow} R\text{CBS}_{\text{TAG}})$, where $L\text{CBS}_{\text{TAG}}$ (left hand side), $K\text{CBS}_{\text{TAG}}$ (left-right gluing), and $R\text{CBS}_{\text{TAG}}$ (right hand side) are TAGs related through graph morphisms l, r. The graph morphisms $lt : L\text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$, $kt : K\text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$ and $rt : R\text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$ ensure that the rule involves only component-based structures. The rule p is said to reconfigure $L\text{CBS}_{\text{TAG}}$ into $R\text{CBS}_{\text{TAG}}$.

Conceptually, a reconfiguration rule encodes a strategy to fulfill a particular QoS level under a given context condition. In our strategy, software evolution architects can take advantage of known design patterns that determine these QoS levels by encoding these patterns in the left and right hand sides of reconfiguration rules. We group all rules to fulfill a given QoS level in the same rule-set to be associated with the respective attribute (*ruleSet*) in a QoS contract instance.

Example 2 (Reconfiguration rules). Table 2 specifies the guaranteeing rule-sets for the QoS contract example on confidentiality for our video conference example. This contract has three QoS levels corresponding to context conditions defined by network access types. Thus, these conditions determine the communications structure of the managed application to guarantee the confidentiality of the transmitted information, following the corresponding design patterns defined in [28, 23, 29]: a clear channel when connected from the intranet; a ciphered channel when from the extranet; and a local cache when having no network access. These reconfiguration rule-sets are defined as follows.

Table 2: Guaranteeing rule-sets for the QoS contract example on confidentiality

Context Events	Expected QoS Level	Guaranteeing Rule Set
1: *from_intranet*	*clearChannel*	*R_clearChannel*
2: *from_extranet*	*confidentChannel*	*R_confidentChannel*
3: *no_Network*	*localCache*	*R_localCache*
Responsibilities		
- System Guarantor:	*System.NetAdapter*[a]	
- Context Monitor:	*System.NetAdapter_AccessPointProbe*[b]	

[a] The component providing the network connection with required QoS conditions.
[b] The component responsible for checking changes on the access points used by the application network connection, and corresponding confidentiality violations.

Rule-set for Changing to Intranet-Serviced Areas (R_clearChannel). Given that the user can move freely among locations with any of the enunciated context conditions, the reconfiguration rules specified to configure a clear channel (i.e., when moving to an intranet-serviced area) must

consider the transitions from any of the two other configurations. Thus, in Figure 8 we present the rule-set to apply when the user moves to an intranet-serviced area from either, an extranet-serviced area (cf. `extra2intra` rule in the top of the figure), or an area having no network access at all (cf. `nonet2intra` rule in the bottom of the figure).

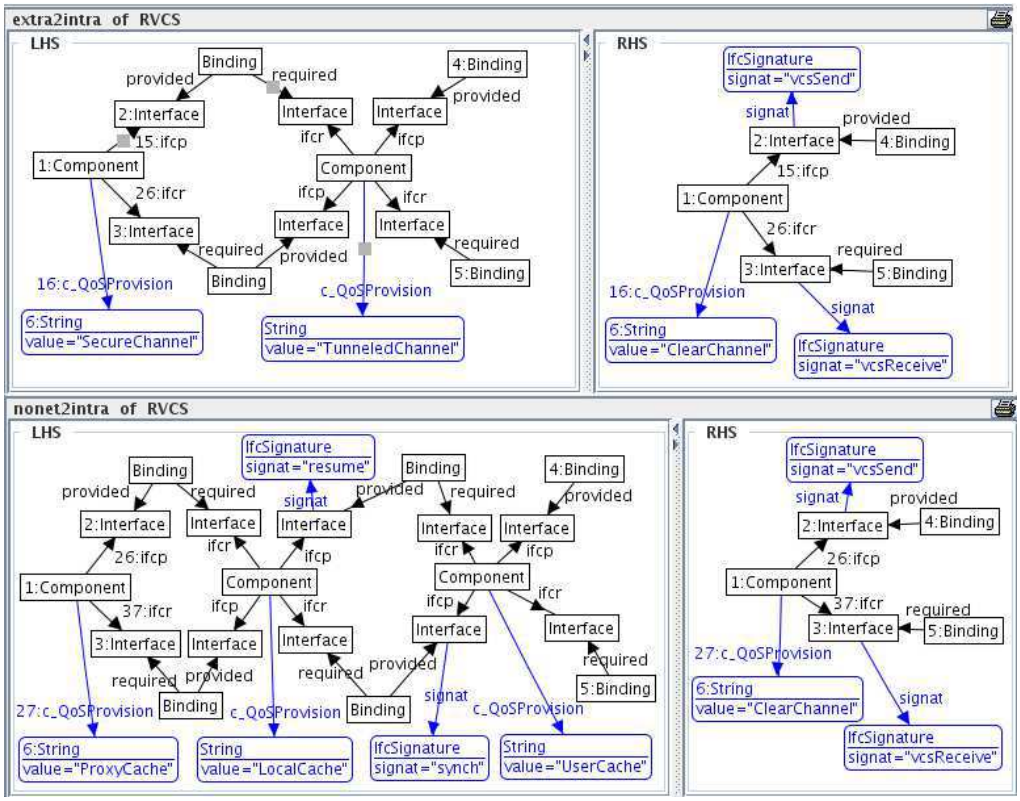

Figure 8: The *R_clearChannel* reconfiguration rule-set.

The left-hand side (LHS) of the `extra2intra` TAG reconfiguration rule is used by pattern-matching to find the components that are providing a *ClearChannel* connection, thus fulfilling the current QoS level (by the *c_QoSProvision* attribute). The right-hand side (RHS) specifies that (i) the matched components by the LHS must be kept with their corresponding bindings as they are, except bindings indexed with 4 and 5; (ii) RHS elements not in the LHS must be removed and undeployed to configure a clear channel; (iii) the existing interfaces indexed with 2 and 3 must be re-bound to bindings indexed with 4 and 5, respectively; finally, (iv) the *c_QoSProvision* attribute of component 1 (i.e., the *NetAdapter*) is updated as provisioning a *ClearChannel*. For reasons of legibility, the left-right gluing $K\text{CBS}_{\text{TAG}}$ and graph morphisms l, r, lt, kt, rt are omitted; $K\text{CBS}_{\text{TAG}}, l, r$ correlate the corresponding LHS-RHS elements, while lt, kt, rt map $L\text{CBS}_{\text{TAG}}$, $K\text{CBS}_{\text{TAG}}$ and $R\text{CBS}_{\text{TAG}}$, respectively, to the $\text{CBSTYPE}_{\text{TAG}}$ structure. The interpretation is similar for the `nonet2intra` rule.

The local-cache design pattern we use in this rule specifies two main components (cf. `LocalCache` in the client-side and `UserCache` in the server-side, bottom rule of the figure) to cope with

communication interruptions. The first provides the asynchronous `resume` service to be used by the second, once the communication is re-established. In complement, the second provides the `synch` service to be used by the `resume` service to synchronize the interactions from both sides, which were saved locally upon the communication interruption, for post-processing. Even though the client- and server-side components must be labeled accordingly in the reconfiguration rules, we have omitted them in this figure for legibility reasons.

Rules for Changing to Extranet-Serviced and Unreachable-Network Areas (R_confidentChannel and R_localCache). From the previously illustrated rules, it is worth noting that in each rule we can invert its LHS and RHS to obtain the rule that specifies the opposite transition between the same pair of states. Moreover, it is easy to observe that we have chosen carefully the same key elements in the LHSs and RHSs (i.e., the `2:Interface`, `3:Interface`, `4:Binding` and `5:Binding`) as pivots for these reconfiguration rules. Thus, the reconfiguration rule-sets to be applied for changing to extranet-serviced areas (*R_confidentChannel* rule-set) and unreachable-network areas (*R_localCache* rule-set) can be specified from the previously specified rule-set, by inverting and exchanging their LHSs and RHSs, that is, with LHS = RHS = {`intranet, extranet,noNetwork`}, as arranged in Figure 9 (e.g., the rule `extranet-to-intranet` is formed with LHS = `extranet` and RHS = `intranet`).

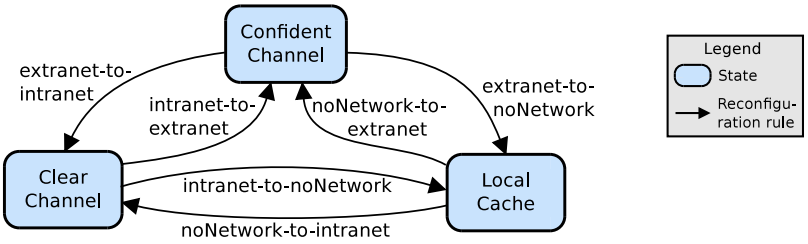

Figure 9: The reconfiguration rule-sets for the confidentiality contract.

Similar to the illustrated reconfiguration rule-set for configuring a clear channel, the corresponding rule-sets to apply when the user moves into an extranet-serviced area, and into an area with no network access, are composed of two reconfiguration rules. These rules correspond to each of the two other possible configurations, as evidenced in the figure, that is, all the states have two incoming transitions.

Definition 6 (Component-Based Structure Reconfiguration System –CBSRS). The component-based structure reconfiguration system, CBSRS, is the tuple ($DSig$, $\text{CBSTYPE}_{\text{TAG}}$, CBSAR, P), where $DSig$ is the data type signature from Def. 2 ($\text{CBSTYPE}_{\text{TAG}}$); $\text{CBSTYPE}_{\text{TAG}}$ the component-based structure typing definition; CBSAR the reflection structure of the managed application to reconfigure in its current running state; and P a set of reconfiguration rules, with $\text{CBSTYPE}_{\text{TAG}}$, CBSAR, and P according to Def. 2, 3, and 5, respectively, for:

1. *(What and Where to reconfigure)* The rule-set P is applied to the managed application reflection structure CBSAR = (CBS_{TAG}, t) by graph-based pattern matching. That is, for each reconfiguration rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ in P, we determine whether there exists a match of p in CBS_{TAG}. To do this, we define the boolean $match(p, \text{CBSAR})$ function, which returns *True* if there exists a morphism $m : L \rightarrow \text{CBS}_{\text{TAG}}$ (called a *matching* of L in the CBS_{TAG}

element of CBSAR), and *False* otherwise. This *matching* determines the CBS_{TAG} potential elements to be reconfigured in the managed application.

2. *(How to reconfigure)* We call a *direct reconfiguration* $G \overset{p,m}{\Rightarrow} G'$ the application of the TAG transformation of G into G', as specified by the reconfiguration rule p, according to Def. 5.
3. (*CBSAR reconfiguration step*) Finally, the CBSAR reconfiguration is performed through the $reconfig(\text{CBSAR}, P)$ operation. This operation is defined as the sequence of direct reconfigurations of the CBS_{TAG} element of CBSAR, $\text{CBS}_{\text{TAG}} = G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n = \text{CBS}_{\text{TAG}}'$, written $\text{CBS}_{\text{TAG}} \overset{*}{\Rightarrow} \text{CBS}_{\text{TAG}}'$, which finishes when no more rules in P can be applied, obtaining a new managed application reflection structure CBSAR′. From the sequence of transformation operations, a correct by construction plan is synthesized to reconfigure the actual running system (i.e., to perform $CBS \rightarrow CBS'$ in the commutative diagram of Section 3.1).

That is, a CBS reconfiguration system reconfigure component-based structure applications. However, as mentioned before, it performs this operation indirectly, by using the guaranteeing reconfiguration rule-sets specified in the QoS contract in the running system denotation, CBSAR, and synthesizing a reconfiguration plan from the graph transformation operations. This reconfiguration plan is finally instrumented in the actual running software system.

Example 3 (System reconfiguration). In Example 1 we presented the component-based structure of the RVCS application when the client is connected from an intranet-serviced area (i.e., with a clear channel). Whenever the user moves into an extranet-serviced area, the reconfiguration rule `intra2extra`, illustrated in Example 2 (i.e., the inverse of `extra2intra`) is applied on the RVCS's component-based structure application reflection state, CBSAR. Figure 10 illustrates the CBS_{TAG} denotation of the reconfigured RVCS application, whose structure fulfills the QoS level for the new context condition (i.e., *confidentChannel* for a network connection *fromExtranet*) as specified in the confidentiality contract. Correspondingly, Figure 11 illustrates the reconfigured runtime application structure in component-based notation.

4.4. Separation of Concerns

In the previous sections we have presented the formal definitions to build our system for reconfiguring component-based applications. In this section we analyze the implications of these definitions.

The most important point to highlight corresponds to the indirect relationships between *QoSProvision* attributes and reconfiguration rules (Def. 2, 3 and 5). *QoSProvision* attributes are used to identify the managed application elements considered as potential reconfiguration objectives. These elements refer to components, interfaces and bindings in both, the managed application reflection structure, as well as left and right hand sides of reconfiguration rules. Thus, the relationships between them are to be discovered at runtime by pattern-matching operations. As a result, these indirect relationships introduce a level of indirection that allows the decoupling of the managed application (to be reconfigured) from the mechanism that actually performs the reconfiguration. In turn, this decoupling allows our model to manage the clear separation of concerns between the contractual QoS properties on the managed application and the properties of our reconfiguration mechanism.

The second point is the central role of the contract definition as the consistency enforcer of our approach with respect to the QoS properties of interest (Def. 4 and 6). This consistency

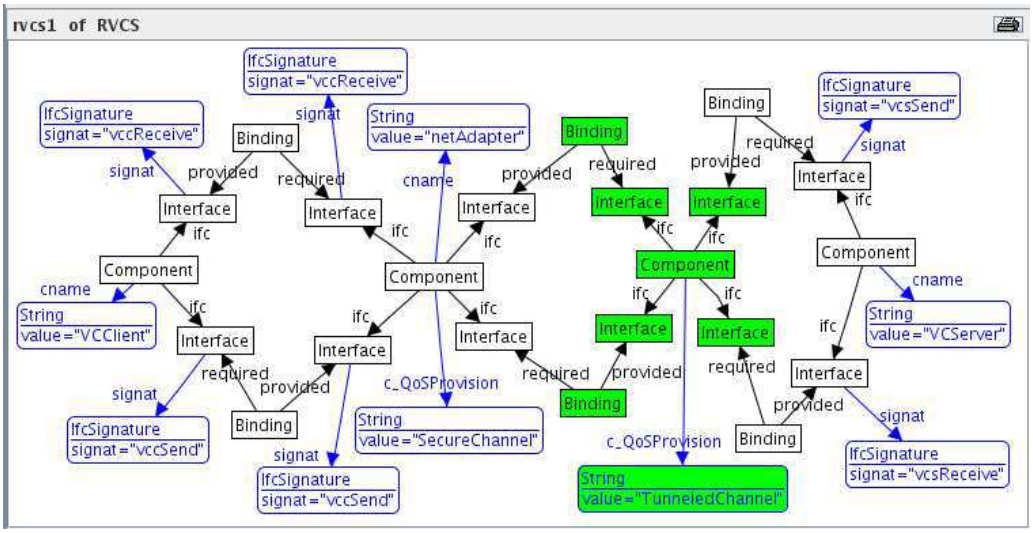

Figure 10: TAG for the reflection system structure reconfigured with rule `intra2extra`. The components that were added and deployed by the applied reconfiguration rule are highlighted.

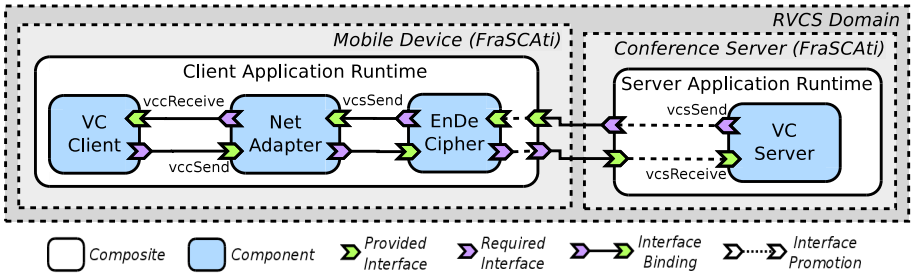

Figure 11: Reconfigured running software structure corresponding to the CBS_{TAG} of Figure 10.

supports the collaborative and coordinated work not only among our denotational elements, but also between them and the elements of the actual running system that participate in the reconfiguration loop, namely the system component monitors and guarantors. In this way, our contract definition serves several objectives: (i) it specifies expected QoS levels of a managed application to its users; (ii) it establishes the responsibilities for the internal components of our approach to fulfill these obligations; and (iii) it declares the responsibilities for the monitor and guarantor elements that would be required to complete a reconfiguration loop in a final deployment.

5. Denotation of QoS Contracts

In Section 3 we introduced the global idea of specifying the semantics of QoS contracts in terms of FSMs, and identified two main challenges for realizing it appropriately. First, the execution of an FSM transition must enforce that the software system subject to a QoS contract fulfills the QoS-level associated to the transition's FSM target state. Second, the FSM representation must guarantee contract robustness with respect to context unpredictability. In Section 4 we

specified the TAG-based denotations for component-based software structures, design-patterns and reconfiguration systems, which define the operational layer of our denotational semantics. This operational layer constitutes the base to solve the first of the two aforementioned challenges. In this section we present the formal definition of the governing layer of our semantics. This layer extends FSMs and we use its elements as denotations for QoS contract elements (i.e., specifically the QoS levels as extended states, context conditions as transition triggers, and guaranteeing rules as transition actions). By parameterizing this second layer with the operational one, we obtain the two-layered denotational semantics for QoS contracts as a whole, which solves both of the two aforementioned challenges in an integrated way.

5.1. $QoSC_{FSM}$: QoS Contract Driven Redefinition of FSMs

The definition of the governing layer of our denotational semantics is based on a redefinition of finite-state machines (FSMs). Even though most of the elements of a QoS contract can be represented with the conventional definition of FSM [38], as illustrated in Section 3, a detailed analysis of the FSM definition allows us to identify several problems with this representation. First, context events must be specified explicitly to label each transition between states. This imposes very strict constraints for the identification and specification of transition triggers—and the transition function (δ) itself—that can lead to a considerable number of transitions [27]. Second, the reconfiguration rules, aimed at guaranteeing QoS levels when triggered by context events, are not considered in FSM transitions. Thus, an explicit invocation mechanism is needed for their execution. Even modifying the FSM to support entry/exit actions as in the event-condition-action strategies [39], the application of our reconfiguration rule-*sets* would require additional mechanisms, as event-condition-actions specify only one action per event-condition. Third, the purpose of QoS contracts is to specify expected QoS levels to be fulfilled under different context conditions. Thus, QoS contracts specify only states of *fulfillment*. However, to cope with robustness, we need to consider also the possible unexpected and undesirable states that could be reached by the managed application when facing context conditions that were not foreseen by the user (e.g., a software evolution architect). Nonetheless, solving these problems is a challenging task as some of the improvements depend on the others, and even more, the improvement of some may imply to worsen the others. For instance, adding unexpected and undesirable states would imply to specify all of the *exact* context events and conditions that may cause the system to transition into these states, which is of course contradictory to the problem of controlling the transitions explosion.

In this subsection and the following we present our solution to the aforementioned problems by taking advantage of the operational layer of denotations defined in section 4. This layer is based on our TAG-based reconfiguration system (CBSRS, Def. 6) as a more powerful way of specifying the FSM states and transitions. To achieve this, we modify the conventional FSM definition in three ways. First, parameterizing it with our CBSRS denotations. Second, adding it with two new elements, and six auxiliary functions. Third, redefining its transition function (δ). The new definition of FSM, QoSC_{FSM}, is as follows.

Definition 7 (QoS-Contract FSM –QoSC_{FSM}). A QoSC_{FSM} is the tuple $\langle STATES, ACCEPT, \Sigma, \Gamma, \Psi, \kappa, \eta, \rho, \mathfrak{R}, \pi, CBSRS, \delta \rangle$, where

- $STATES$ is the finite set of expected QoS levels to fulfill. As these levels are expressed as predicates, we label each state with the corresponding indexed predicate.

- $ACCEPT \subseteq STATES$ is the set of states of contract fulfillment. The final managed application state, once finished its execution, should correspond to one of these states.
- Σ (the controlled context events alphabet) is the finite set of context events specified by the user to notify changes of QoS levels, thus requiring the managed application to be reconfigured. In contrast, we distinguish this set from Σ^U, the set of all possible context events that can be reported from context monitors.
- Γ is the set of guaranteeing rules, specified according to our definition of CBS reconfiguration rules (Def. 5). These rules specify how to perform reconfigurations (i.e., transitions) between the graph representations of running-software states.
- $\Psi : STATES \rightarrow PREDICATE$ maps each target state with the predicate (i.e., QoS level) to be fulfilled on it. $PREDICATE$ is the set of well-formed first-order logic formulae.
- $\kappa : STATES \rightarrow PREDICATE$ maps each target state with the corresponding context condition that holds on it.
- $\eta : STATES \rightarrow \mathcal{P}(\Sigma)$ maps each target state with the type of the context events that trigger transitions to it. $\mathcal{P}(\Sigma)$ is the power set (i.e., the set of all subsets) of Σ.
- $\rho : STATES \rightarrow \mathcal{P}(\Gamma)$ maps each target state to the guaranteeing reconfiguration *rule-set* to fulfill its QoS-level objective.
- $\mathfrak{R} : STATES \rightarrow CBSAR$ maps the actual contract state to the corresponding running-software state (i.e., the Component-Based Software Application Reflection structure, CB-SAR, Def. 3). As the managed application may evolve dynamically from state to state, its structure is not necessarily the same for a same state. Thus, this mapping is not statically defined, but computed at runtime only for the current state.
- $\pi : STATES \rightarrow QoSProperty$ maps each target state to the QoS property in which it is defined. $QoSProperty$ is the finite set of QoS properties for which the user specifies context events, QoS levels, and reconfiguration rules.
- $CBSRS$ is the component-based structure reconfiguration system (from Def. 6).
- $\delta : \Sigma \times \Delta^U \rightarrow \{Fulfilled, Unstable, Exception\}$ is the *transition function*, where "Fulfilled", "Unstable", and "Exception" are super-states grouping the respective states, and Δ^U is a reference to the universal execution context space of software applications. This space is left unspecified intentionally, given the practical impossibility of formalizing it, even in an abstract form. Any abstraction would imply a partial representation, which, at some point would turn to be incomplete and formally unsound. Instead of that, managing it as a reference to an external construct allows us to refer to its particular instances when evaluating the managed application by sensing the actual context of execution, which is more realistic and feasible than formalizing it.

In the following subsections we illustrate the elements of this FSM redefinition and how it solves the aforementioned problems.

5.1.1. The $QoSC_{FSM}$ Auxiliary Functions

Given that transitions in QoSC_{FSM} are driven by context events and system reconfiguration rules, we encode this information in five of the added auxiliary functions: Ψ, κ, η, ρ and π. $\mathfrak{R}$, the sixth function, is used to map the current QoSC_{FSM} state with the running application state. According to their definitions, $\eta : STATES \rightarrow \mathcal{P}(\Sigma)$ can be used to identify the set of triggering

events that cause a transition to a given state; and vice versa, given a context event, to identify the target state to which this event triggers a transition; $\Psi : STATES \rightarrow PREDICATE$ to identify the predicate expressing the QoS level that must be fulfilled in a given state; and $\rho : STATES \rightarrow \mathcal{P}(\Gamma)$ to retrieve the set of reconfiguration rules to be applied for fulfilling the QoS level associated to a given state. $\kappa : STATES \rightarrow PREDICATE$ is used to obtain the context condition associated with a target state, for instance to confirm if it is still in force in the current context situation, and $\pi : STATES \rightarrow QoS\,Property$ to determine the QoS property to which a given state belongs to.

The mappings for Ψ, κ, η, ρ, and π can be automatically obtained from QoS contract instances. For the contract example on confidentiality, corresponding to the QoS level to be fulfilled when moving into an extranet-serviced area, these mappings are the following[4]:

- $\Psi(confidentChannel) \mapsto$ "$confidentChannel$"
- $\eta(confidentChannel) \mapsto from\_extranet$, with $from\_extranet = \{FromExtranet\}$
- $\rho(confidentChannel) \mapsto R\_confidentChannel$, with $R\_confidentChannel$ the set of two reconfiguration rules {`intranet-to-extranet`, `noNetwork-to-extranet`} (cf. rules from Example 2)

Here we present the values of Ψ, which represent predicates, as strings. These predicates are evaluated in the execution of the reconfigured managed application under the current context of execution by the function *evalInContext*. The values returned by this function are (F)ulfilled and (U)nfulfilled. The purpose of *evalInContext* is to verify, by sensing the execution context at its invocation time, whether the predicate that corresponds to a given QoS level is satisfied (returning F), or not (returning U). In this example, $\Psi(confidentChannel)$, which is mapped to the predicate "$confidentChannel$", would be evaluated by sensing if the (re)configured application fulfills the predicate confidentChannel (i.e., the transmitted data is wrapped in a ciphered packet). In the case of a contract on throughput, $\Psi(T100/min)$ could be mapped to the predicate "$T100/min$", which would be evaluated by sensing if the application is performing 100 transactions per minute at invocation time. This function is fundamental in self-adaptive systems, given that it is not obvious that the condition will hold after performing a reconfiguration. Of course, to cope with this problem, the user must provide relevant corresponding reconfiguration rules.

In the case of the confidentiality example, assume that we have a QoSC_{FSM} with the function mappings defined as above. We would also have $\Sigma = \{FromIntranet, FromExtranet, NoNetwork\}$ and $\Gamma = \{R\_clearChannel, R\_confidentChannel, R\_localCache\}$. When the current state of the managed application is connected from the intranet and changes to an extranet-serviced area, it receives the event $e = FromExtranet$, and, of course, $e \in \Sigma$ holds. Thus, we have:

- $e = FromExtranet \in \eta(confidentChannel)$ holds, given that $\eta(confidentChannel) \mapsto \{FromExtranet\}$.
- The target state s, into which the FSM must make a transition having occurred the event e, is obtained with the formula $\exists s : s \in STATES \mid e \in \eta(s)$, that is, $s = confidentChannel$.
- The QoS level to fulfill in this target state s is $\Psi(\exists s : s \in STATES \mid e \in \eta(s))$, that is, $\Psi(confidentChannel) \mapsto$ "$confidentChannel$".
- The rule-set that must be applied to fulfill the QoS level implied by this target state s is $\rho(\exists s : s \in STATES \mid e \in \eta(s))$, that is, $\rho(confidentChannel) \mapsto$ {`intranet-to-extranet`,

[4]Despite any names could be used, we chose very representative ones to make the examples straightforward to read.

noNetwork-to-extranet}.

5.1.2. Managing Exception and Unstable States of Contract Unfulfillment

So far, our interpretation of QoS contracts is based on the expected QoS levels to fulfill, as specified in QoS contract instances. This implies that up to this point our model considers only the states of contract fulfillment. However, from the analysis of the auxiliary functions it is clear that to address contract robustness facing situations unforeseen by the user and derived from unexpected context conditions, contract *unfulfillment* states must be also managed. These states, to be added to the states of contract fulfillment in their automatic derivation from QoS contract instances, are the following:

- The *exception* state, modeling the specific situations in which the user specifies either:
  - an incomplete set of reconfiguration rules (i.e., no rule matches a given managed application state or context condition); or
  - a set of reconfiguration rules whose application results in a managed application that violates the component-based integrity constraints; or
  - a set of context events excluding others that actually occur in the application execution context.
- The *unstable* state, which models the situation in which the specified rules are not relevant or not enough to achieve the fulfillment of a given QoS level.

The value of these states, besides the user not having to worry about robustness issues such as specifying undesirable states and transitions, is that we can define the formal semantics of these states and transitions systematically. In this way, different types of operational actions to warn the user at runtime about the corresponding anomalous situations can be associated to these states, for instance after a repeated number of occurrences.

5.1.3. Generalizing Conditions for the $QoSC_{FSM}$ Transitions

Based on the definition and analysis of the $\mathrm{QoSC_{FSM}}$ structure and its auxiliary functions, we can notably generalize the conditions for all of the transitions to the states of contract fulfillment. To achieve this, besides the *evalInContext* function, we recall the *CBSRS* boolean functions *match* and *reconfig* that we defined with it (cf. Section 4.3). $match(r, \mathrm{CBSAR})$ returns true if there exists a match of the LHS of rule r in CBSAR, and false otherwise. Having a reconfiguration rule-set γ such that $r \in \gamma$ and $match(r, \mathrm{CBSAR})$ holds, $reconfig(\mathrm{CBSAR}, \gamma)$ uses γ to reconfigure CBSAR, the managed application reflection structure. If the reconfiguration succeeds, that is, the application of the rule-set produces a managed application reflection structure that conforms to the component-based structural constraints, the function returns true; otherwise, it returns false.

Thus, the general condition for all of the transitions to contract fulfillment states given the context event e, under the execution context Δ, is:

$$\begin{aligned} &e \in \Sigma \,\overline{\wedge}\, \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s))[match(r, \Re(s \,|\, e \in \eta(s)))\overline{\wedge} \\ &reconfig(\Re(s \,|\, e \in \eta(s)), \rho(s \,|\, e \in \eta(s))) \,\overline{\wedge}\, evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F] \end{aligned} \tag{1}$$

We read this condition as follows:

- given the occurrence of event e (i.e., a new context situation is present), considered among the set of valid context events Σ, and
- a target state s to be reached in the new context signaled by e (i.e., $s \,|\, e \in \eta(s)$), and

- a reconfiguration rule r in the rule-set to guarantee the fulfillment of the QoS level associated to s (i.e., $r \in \rho(s \mid e \in \eta(s))$) such that we can find a match of r in the current graph structure of the managed application $\mathfrak{R}(s \mid e \in \eta(s)))$, and
- after reconfiguring the application reflection structure with the respective rule-set $(reconfig(\mathfrak{R}(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s))))$, the QoS level associated to the target state is F (Fulfilled). That is, $evalInContext(\Psi(s \mid e \in \eta(s)), \Delta) = F$.

It is worth noting that the occurrence of matching rules in the managed application upon a change in context conditions necessarily implies a reconfiguration. We use the symbols $\overline{\wedge}$ and $\overline{\vee}$ for the *consecutive logical* conjunction and disjunction, respectively. Consecutive expressions operated with the $\overline{\wedge}$ ($\overline{\vee}$) operator are evaluated from left to right and stops in the first one that evaluates to false (true), if any, being it an operational version of logical conjunction (disjunction) used in the theory of abstract syntax-directed translation [40].

In condition (1) we highlight the predominance of *target states* over source states that results from the context-driven nature of these transitions. Thus, for each of the target states, which correspond to the QoS levels to be fulfilled, we only need to specify the conditions required by the respective *incoming* transitions. The justification for this is that we use the LHSs of reconfiguration rule-sets as part of the condition for the incoming transition to be satisfied in the *match* operation. In this way, in the generalized condition (1) we capture the transitions that the user may specify coming from any of the possible source states (i.e., all other QoS levels specified in the contract). That is, the LHSs determine which reconfiguration rules apply.

Furthermore, we follow this same strategy for the added states of contract unfulfillment (*Unstable* and *Exception*). For the transitions into these two states we need similar conditions, keeping in mind that those states result from anomalous situations encountered when trying to reach fulfillment states. These conditions correspond to the robustness requirements given in Section 5.1.2. Hence, the condition to reach the *unstable* state is the same as (1), except that the QoS level to be fulfilled in the target state is not achieved (i.e, the QoS-level predicate is evaluated as (U)nfulfilled), meaning that the rules given by the user are not relevant or not sufficient to achieve the fulfillment of the QoS level:[5]

$$\begin{aligned} &e \in \Sigma \,\overline{\wedge}\, \exists s, r : s \in STATES, r \in \Gamma \mid r \in \rho(s \mid e \in \eta(s))[match(r, \mathfrak{R}(s \mid e \in \eta(s)))\overline{\wedge} \\ &reconfig(\mathfrak{R}(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s)))\,\overline{\wedge}\, evalInContext(\Psi(s \mid e \in \eta(s)), \Delta) = U] \end{aligned} \tag{2}$$

For the *exception* state we have three disjoint conditions. The first expresses that the user specified an incomplete set of reconfiguration rules (i.e., no rule matches a given application state or context condition: $\forall r : r \in \rho(s \mid e \in \eta(s))[\neg match(r, \mathfrak{R}(s \mid e \in \eta(s)))]$); the second, that an event occured but it is not in the set of context events specified by the user as valid ($e \in \Sigma^U \setminus \Sigma$); whereas the third, that the application of some of the rules given by the user violate the structural integrity constraints of component-based software (i.e., $\neg reconfig(\mathfrak{R}(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s))))$:

$$\begin{aligned} &e \in \Sigma \,\overline{\wedge}\, \exists s : s \in STATES \mid \forall r : r \in \rho(s \mid e \in \eta(s))[\neg match(r, \mathfrak{R}(s \mid e \in \eta(s)))]\vee \\ &(e \in \Sigma^U \setminus \Sigma)\vee \\ &(e \in \Sigma \,\overline{\wedge}\, \exists s, r : s \in STATES, r \in \Gamma \mid r \in \rho(s \mid e \in \eta(s))[match(r, \mathfrak{R}(s \mid e \in \eta(s)))\overline{\wedge} \\ &\neg reconfig(\mathfrak{R}(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s)))]) \end{aligned} \tag{3}$$

[5] Again, e is the particular event notified by a context monitor. Thus, it is bound to the respective sensed value.

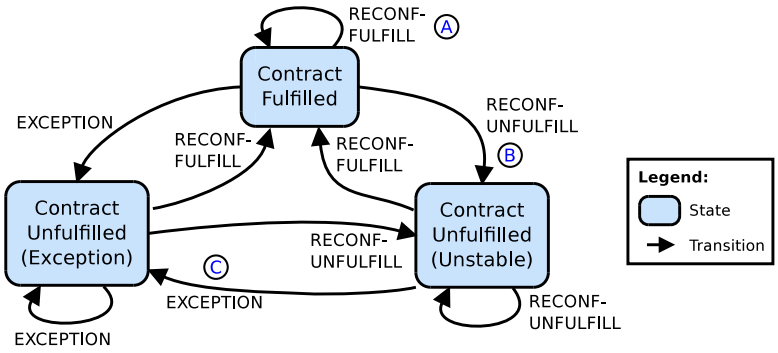

Figure 12: State machine for the QoS contract-preserving reconfiguration system.

5.1.4. The Transition Function

From the previous analysis, it is easy to observe that the same condition (1) expresses correctly the transition requirements that target the three states of contract fulfillment of the confidentiality example. Moreover, based on this observation, we can abstract all the FSM fulfillment states as one "Contract Fulfillment" super-state to simplify the presentation of QoSC_{FSM}s without loss of generality.[6] In this way, our semantic denotation for QoS contracts preservation becomes the state-machine depicted in Figure 12. In this figure, three possible transitions, corresponding to the generalized transition conditions (1), (2) and (3), may occur: Ⓐ RECONF-FULFILL: if the QoSC_{FSM} finds a matching reconfiguration rule and performs a reconfiguration on the application, verifying that the reconfigured application fulfills its obligations; Ⓑ RECONF-UNFULFILL: this transition operates as in (1), except that after the reconfiguration, the QoSC_{FSM} verifies that the reconfigured managed application is not yet fulfilling the respective QoS level, thus reaching an *unstable* yet contract-unfulfillment state; Ⓒ EXCEPTION: being unable to find a matching rule to apply, or finding and applying a matching rule that produces an invalid reconfiguration, or facing an unspecified context event, the QoSC_{FSM} is left in an *exception* state.

Based on the three target super-states, *Fulfilled*, *Unstable*, and *Exception*, and their generalized transition conditions, we define our QoSC_{FSM} transition function as follows.

Definition 8 (QoSC_{FSM} Transition Function). The transition function $\delta : \Sigma \times \Delta^U \rightarrow \{Fulfilled, Unstable, Exception\}$ (cf. Def. 7) defines output states depending on the context event (e) received from context monitors, and the execution context (Δ):

[6] The abstraction of the "fulfillment" state can be seen as a superstate in the sense of Harel statecharts, in which substates can transition internally without affecting other states in the statechart. Harel statecharts are used to improve the legibility of finite-state machine representations [41]. Besides concurrency, this formalism models clustering, hierarchy and history with an excessively complex semantics, none of this required in our model.

$$\delta(e,\Delta) = \begin{cases} \text{Fulfilled}^* & \text{if } e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \mid r \in \rho(s \mid e \in \eta(s))[match(r, \Re(s \mid e \in \eta(s))) \barwedge \\ & \quad reconfig(\Re(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s))) \barwedge evalInContext(\Psi(s \mid e \in \eta(s)), \Delta) = F] \\ \text{Unstable} & \text{if } e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \mid r \in \rho(s \mid e \in \eta(s))[match(r, \Re(s \mid e \in \eta(s))) \barwedge \\ & \quad reconfig(\Re(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s))) \barwedge evalInContext(\Psi(s \mid e \in \eta(s)), \Delta) = U] \\ \text{Exception} & \text{if } e \in \Sigma \barwedge \exists s : s \in STATES \mid \forall r : r \in \rho(s \mid e \in \eta(s))[\neg match(r, \Re(s \mid e \in \eta(s)))] \barvee \\ & \quad (e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \mid r \in \rho(s \mid e \in \eta(s))[match(r, \Re(s \mid e \in \eta(s))) \barwedge \\ & \quad \neg reconfig(\Re(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s)))]) \barvee (e \in \Sigma^U \setminus \Sigma) \end{cases}$$

For the target *Fulfilled** state, $s \mid e \in \eta(s)$ is the actual QoS-level state of fulfillment, while $\Psi(s \mid e \in \eta(s))$ the corresponding QoS-level predicate to be satisfied. Σ^U, as defined previously, is the set of all possible context events that can be reported from context monitors.

5.1.5. Automatic Synthesis of $QoSC_{FSM}$

To take full advantage of QoSC_{FSM} in practical terms (i.e., in actual running software systems), our denotational semantics for QoS contracts must be executable. Algorithm 1 obtains the QoSC_{FSM} denotation from a QoS contract instance by creating the states of contract fulfillment, adding the states of contract unfulfillment, and defining the mappings for the auxiliary functions and structures. These instances are specified in XML files using Eclipse or the AGG software tool with the $\text{CBSTYPE}_{\text{TAG}}$ definition as document type. Thus, this algorithm produces correct-by-construction QoSC_{FSM} with respect to the QoS contract typing specification.

5.2. $QoSC_{FSM}$ Execution

We characterize the execution state of the QoSC_{FSM} as a binary relation (i.e., a vector) between QoS properties and respective QoS levels.[7]

Definition 9 (QoSC_{FSM} Execution State –QES). Given QoSC a QoS contract instance, and CBSAR the component-based structure application reflection state subject to QoSC, the QoSC_{FSM} execution state is the tuple $\langle CURSTATE, \text{CBSAR} \rangle$, where CURSTATE is the binary relation $QoSC.property \times (QoSC.property.obligation.SLOPredicate \cup \{Exception, Unstable\})$ such that each QoS property appears exactly once in these pairs, related either to one of its corresponding QoS levels, or to the exception or unstable state.

That is, $CURSTATE$ is defined by the set of pairs (QoS-property, QoS-level) for each QoS property in the contract. Each pair relates a QoS property either to its current QoS level state, if this is fulfilled by the managed application, or to the exception or unstable state, if unfulfilled. Algorithm 2 illustrates the QoSC_{FSM} execution control block, which is executed in response to reconfiguration context events.

In the RVCS system of our application scenario, the contract defines a QoS property of confidentiality (cf. Table 1a). Thus, $\pi = \{(clearChannel, Confidentiality), (confidentChannel,$

[7] Even though QoSC_{FSM} can manage multiple QoS properties in a QoS contract, in this paper we leave this aspect out of discussion because of space limitations.

Algorithm 1 QoSC_{FSM}_from_QoS_Contract

Input: $qosContract : QoSC$ /* QoSC from Def. 4 */
Output: $qfsm : \langle STATES, start, ACCEPT, \Sigma, \Gamma, \Psi, \kappa, \eta, \rho, \Re, \pi, CBSRS \rangle$ /* QoSC_{FSM}, Def. 7 */

1: Initialize $STATES, \Sigma, \Gamma, \Psi, \kappa, \eta, \rho, \pi$ with { }
2: **for all** $qosProp \in qosContract.C.property$ **do**
3: **for all** $sloOblig \in qosProp.obligation$ **do**
4: $curState \leftarrow make\_indexed\_state(sloOblig.SLOPredicate)$
5: $qfsm.STATES \leftarrow qfsm.STATES \cup \{curState\}$/* build all the FSM states */
6: $qfsm.\Sigma \leftarrow qfsm.\Sigma \cup sloOblig.contextEvType$ /* the FSM events */
7: $qfsm.\Gamma \leftarrow qfsm.\Gamma \cup sloOblig.ruleSet$ /* and the FSM reconfiguration rules */
8: $qfsm.\Psi \leftarrow qfsm.\Psi \cup \{(curstate, sloOblig.SLOPredicate)\}$/* represented predicate */
9: $qfsm.\kappa \leftarrow qfsm.\kappa \cup \{(curstate, sloOblig.contextCondition)\}$/* context condition */
 /* Then, for the FSM transitions: index and associate: */
10: $eventSet \leftarrow make\_indexed\_set(sloOblig.contextEvType)$ /* the events */
11: $ruleSet \leftarrow make\_indexed\_set(sloOblig.ruleSet)$ /* and associated ruleSet */
12: $qfsm.\eta \leftarrow qfsm.\eta \cup \{(curstate, eventSet)\}$
13: $qfsm.\rho \leftarrow qfsm.\rho \cup \{(curstate, ruleSet)\}$
14: $qfsm.\pi \leftarrow qfsm.\pi \cup \{(curstate, qosProp)\}$
15: **end for**
16: **end for**
17: $qfsm.ACCEPT \leftarrow qfsm.STATES$
18: $qfsm.STATES \leftarrow qfsm.STATES \cup make\_indexed\_state(Exception)$
19: $qfsm.STATES \leftarrow qfsm.STATES \cup make\_indexed\_state(Unstable)$
20: **return** $qfsm$

Algorithm 2 QoSC_{FSM} transition executor

Input: $e : \Sigma^U, qes : QES, qfsm : \text{QoSC}_{\text{FSM}}$ /* QES from Def. 9, QoSC_{FSM} from Def. 7 */
Output: qes transitioned on the property affected by e, using δ (Def. 8)

1: $S' \leftarrow \delta(e, qfsm.\Delta)$
2: $qes.CURSTATE[\pi(s \mid e \in qfsm.\eta(s))] \leftarrow S'$
3: **if** $S' == Exception$ **then** /* reconfiguration problems */
4: qes.CBSAR $\leftarrow getCBSAR()$ /* recover the previous state of CBSAR */
5: **else** /* CBSAR successfully reconfigured in advance w.r.t. the managed application */
6: instrument_reconfiguration_plan_in_runtime_platform()
7: **end if**
8: **return** qes

$Confidentiality$), ($localCache$, $Confidentiality$)}. When initially executed from an intranet-serviced area, the QoSC_{FSM} execution state is $CURSTATE = \{(Confidentiality, clearChannel)\}$. Then, when the software client moves into an extranet-serviced area, the QoSC_{FSM} executor is invoked. Line 1 performs the state transition using the δ function and reconfigures CBSAR. Line 2 updates the QoSC_{FSM} execution vector state (i.e., $CURSTATE$) on the property affected by e (i.e., $\pi(s \mid e \in qfsm.\eta(s))$), with the state returned by δ. We use the notation $qes.CURSTATE[p] \leftarrow s$ to express that the QoS property p is associated with s in the relation $CURSTATE$. Hence, the execution state transitions into $CURSTATE = \{(Confidentiality, confidentChannel)\}$.

However, given that the reconfiguration operates on CBSAR and verifies its component-based structural conformance before instrumenting the respective changes in the managed application, CBSAR is left inconsistent when this verification fails (i.e., the transition resulted in the

Exception state as a result from the application of a faulty reconfiguration rule). In this case, line 4 recovers the previous state of CBSAR from the running managed application. Otherwise, the changes in the CBSAR are instrumented in the running system.

Finally, as the reconfiguration depends on the matching operation between reconfiguration rules and the managed application reflection structure, maintaining CBSAR updated allows further reconfiguration cycles in the managed application. In effect, even if it reaches an *Exception* state, as the managed application is left unmodified by the reconfiguration mechanism, this is not an impediment for its services to continue with their operation, although in a contract unfulfillment state. On the next context event, as CBSAR reflects the current state of the managed application, and thanks to our generalized conditions for transitions to *target* states, a reconfiguration cycle can be performed with no special considerations.

5.3. The QoS Contract-Preserving Reconfiguration System

As we mentioned previously, the final objective of the previous definitions is the autonomous and reliable preservation of QoS contracts under varying context conditions. Contract preservation is defined in terms of continuous reconfiguration cycles that start and end in actual running software applications, triggered by context changes, such that the software applications fulfill the expected QoS levels under the different context situations. To complete the realization of our QoSC_{FSM} as the semantics of our QoS contract definition, we define our QoS contract-preserving reconfiguration system, based on the previous definitions.

Definition 10 (QoS Contract-Preserving Reconfiguration System –QoSCRS). The QoS contract preserving reconfiguration system, QoSCRS, is the tuple $\langle \text{CBSAR}, \mathit{QoSC}, \mathit{CBSRS}, \text{QoSC}_{\text{FSM}} \rangle$, where CBSAR is the reflection structure of the managed application subject to the contract *QoSC*; *CBSRS* the component-based reconfiguration system; and QoSC_{FSM} the state-machine for *QoSC*, all of them according to their respective definitions. On this tuple, we define a QoS contract-preserving reconfiguration cycle as:

1. *(When to reconfigure)* A managed software application reconfiguration is triggered whenever the *QoSMonitor* specified in the contract, *QoSC.monitor*, notifies a context event e that challenges the fulfillment of the current QoS level.
2. *(What, Where and How to reconfigure)* From the contract denotation QoSC_{FSM} and based on the context event e received, the affected QoS property (i.e., $\pi(s \mid e \in \eta(s))$) is identified. Then δ, the transition function is invoked, identifying the target state to be reached under the new context situation (i.e., $s \mid e \in \eta(s)$). Also, the corresponding QoS-level predicate and guaranteeing reconfiguration rule-set for this state are retrieved (i.e., $\Psi(s \mid e \in \eta(s))$ and $\rho(s \mid e \in \eta(s))$, respectively). With this, a state transition in the managed application reflection structure is induced, thus performing a reconfiguration $\text{CBSAR} \overset{*}{\Rightarrow} \text{CBSAR}'$ using the component-based reconfiguration system (i.e., $\mathit{reconfig}(\text{CBSAR}, \rho(s \mid e \in \eta(s)))$). From this, a reconfiguration plan is synthesized (cf. Def. 6).
3. *(Pre-update checks)* Once obtained CBSAR′, the component-based structural conformance check is performed on it. We specify the corresponding conditions in Section 6.1. The verification of these conditions is performed in the *reconfig* function, and hence, their violation would lead to a contract unfulfillment state (with the respective notification to the user).
4. *(Managed-application reconfiguration update)* If the new managed application reflection structure CBSAR′ satisfies the pre-update checks, the synthesized reconfiguration plan is

applied to the running managed application. Otherwise, the managed application is left without modifications, and the user notified. In any case, both, the contract state and the running-software state are updated in consequence, in the QoSC_{FSM} execution state. In this way, the atomicity property is guaranteed, as detailed in Section 6.2.

Given the generic conditions that we established in the semantics of our QoSC_{FSM} *incoming* transitions for target states, the user must only specify the QoS levels (i.e., the FSM states) with the required guaranteeing reconfiguration rules (i.e, the *incoming* transitions for the specified states). That is, the user implicitly select the source states of the transitions by encoding the patterns associated to these states in the LHS of the reconfiguration rules.

6. QoS-CARE Reconfiguration Properties

In this section we analyze the properties of our QoS contract-preserving reconfiguration system, as a realization of the formal semantics presented previously.

6.1. Component-Based Structural Conformance

This property is defined as the conformance of the managed application with respect to the structural integrity constraints defined by the component-based (SCA) specification. This property must be preserved after each reconfiguration.

Definition 11 (Full CB-Structural Conformance). A runtime system reflection structure, CBSAR, is *full CB-structural conformant* iff its CBS_{TAG} is a *component-based structure* (i.e., there exists a graph morphism $t : \text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$), and the following conditions hold:

1. $\forall b1, b2((b1, b2 \in \text{CBS}_{\text{TAG}}.Binding \wedge b1.provided = b2.provided \wedge b1.required = b2.required) \implies b1 = b2)$: every binding must connect different pairs of provided-required services.
2. $\forall b \exists i, j(b \in \text{CBS}_{\text{TAG}}.Binding \wedge i, j \in \text{CBS}_{\text{TAG}}.Interface \wedge i \neq j \wedge b.provided = i \wedge b.required = j \wedge i.signat = j.signat)$: each binding connects different services that, nonetheless, have the same interface.
3. $\forall i(i \in \text{CBS}_{\text{TAG}}.Interface \implies \exists c(c \in \text{CBS}_{\text{TAG}}.Component \wedge (c.ifcp = i \vee c.ifcr = i)))$: every service must belong to some component.
4. $\forall i \exists c((i \in \text{CBS}_{\text{TAG}}.Interface \wedge c \in \text{CBS}_{\text{TAG}}.Component \wedge c.ifcr = i) \implies \exists b, j, d(b \in \text{CBS}_{\text{TAG}}.Binding \wedge b.required = i \wedge b.provided = j \wedge j \in \text{CBS}_{\text{TAG}}.Interface \wedge d \in \text{CBS}_{\text{TAG}}.Component \wedge d.ifcp = j \wedge c \neq d))$: all services required by a component must be bound to services provided by another component.

The verifiability of full CB-structural compliance naturally results from Def. 2 and 3. Referred to our example scenario, it is straightforward to verify that the specified conditions are satisfied by the system reflection structures presented in Figure 6 and 10.

6.2. Atomic Reconfiguration

Atomicity in self-adaptive software systems is the property such that, either the reconfiguration (adaptation) process finishes successfully and the system is reconfigured, or it fails and the system preserves its (previous) configuration and state. Our proof for this property is as follows.

Our strategy to apply design patterns to preserve QoS contracts is based on component-based software reconfiguration. Nonetheless, following steps 1 and 2 of our QoS Contract-Preserving

Reconfiguration System (Def. 10), we first perform the reconfiguration on a graph model of the actual managed application and obtain a reconfiguration plan (Def. 6). In step 3 the reconfigured graph model is verified in its *Full CB-Structural Conformance*, and in step 4 the actual managed application is reconfigured by instrumenting the reconfiguration plan in it. In addition, the *Full CB-Structural Conformance* is the only verification defined and managed in our semantic model that could make the reconfiguration fail.

Therefore, given that QoS-CARE performs a reconfiguration in the actual running managed application only if the *Full CB-Structural Conformance* verification succeeds, it is easy to conclude that QoS-CARE guarantees the atomicity of the reconfiguration process.

6.3. Reconfiguration Termination

Heckel *et al.* and Ehrig *et al.*, among others, showed several graph transformation theorems and results on termination conditions as valid also for typed attributed graph transformation systems (TAGTS) in [42], [43], and [36]. In this section, we show that the *CBS reconfiguration step* of our component-based structure reconfiguration system (i.e., $G_0 \overset{*}{\Rightarrow} G_n$ in CBSRS, Def. 6), is reducible to a typed attributed graph transformation system, as defined in [36].

Theorem 1 (Reducibility of CBS Reconfiguration Step)**.** *Let $CBSRS$ be a component-based structure reconfiguration system according to Def. 6. A CBS reconfiguration step in $CBSRS$ is reducible to a typed attributed graph transformation system, $TAGTS$.*

Proof. According to Def. 6, a $CBSRS$ is a tuple $(DSig, \text{CBSTYPE}_{\text{TAG}}, \text{CBSAR}, P)$. Of these elements, during the *CBS reconfiguration step* (i.e., $G_0 \overset{*}{\Rightarrow} G_n$), the data signature, $DSig$, the component-based structure typing definition, $\text{CBSTYPE}_{\text{TAG}}$, and the set of reconfiguration rules P remain unchanged. Therefore, in a *CBS reconfiguration step* these elements can be omitted, depending only on the system reflection structure, CBSAR, and the set of reconfiguration rules, P. Given that

1. a Component-Based Structure Application Reflection (CBSAR, Def. 3) is a tuple $(\text{CBS}_{\text{TAG}}, t)$, where CBS_{TAG} is the TAG that denotes the component-based structure of the managed system CBS through the one-to-one function $f : \text{CBS} \rightarrow \text{CBS}_{\text{TAG}}$, and t is the TAG morphism $t : \text{CBS}_{\text{TAG}} \rightarrow \text{CBSTYPE}_{\text{TAG}}$, hence in the *CBS reconfiguration step*, the definition of f remains also unchanged; and
2. a typed attributed graph is a tuple (AG, u), where AG is an attributed graph over a data signature $TAGDSig$, and u is an attributed graph morphism, $u : AG \rightarrow ATG$, where ATG is a type graph; and
3. a component-based reconfiguration rule, p, according to Def. 5, is a tuple $(L, K, R, l, r, lt, kt, rt)$, $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, with $lt : L \rightarrow \text{CBSTYPE}_{\text{TAG}}$, $kt : K \rightarrow \text{CBSTYPE}_{\text{TAG}}$ and $rt : R \rightarrow \text{CBSTYPE}_{\text{TAG}}$; and
4. the typed attributed graph transformation rules are graph rewriting productions $q = (X \overset{x}{\leftarrow} Y \overset{y}{\rightarrow} Z)$, X, Y, Z graphs; and
5. both, the TAG system reflection structure CBS_{TAG} and the typed attributed graphs are based on the same TAG definition,

then, a *CBS reconfiguration step* can be reduced to a typed attributed graph transformation system, $TAGTS$, by making $TAGDSIG = DSig$, $AG = \text{CBS}_{\text{TAG}}$ and $ATG = \text{CBSTYPE}_{\text{TAG}}$. The $TAGTS$ set of transformation rules can be defined as the set of component-based reconfiguration

rules without the lt, kt, rt morphisms, given that, once defined the component-based reconfiguration rules, these morphisms are also unchanged. □

As a result, the aforementioned theorems on termination properties are also valid for our reconfiguration system. In particular, we use the criteria established by Taentzer and others in [35, 44, 45] to determine whether the *CBS reconfiguration step* with the user-defined reconfiguration rules in our reconfiguration system is terminating, contributing to its reliability.

6.4. Contract Robustness

Contract robustness with respect to context unpredictability requires guaranteeing the delivery of the agreed managed software application services on any of the managed application states, that is, even in states that differs from the expected ones. These states are those defined in the QoS contract with respect to the context situations, as specified by the user. Therefore, to show that QoS-CARE guarantees robustness, it is sufficient to show that the QoS-CARE model, QoSC_{FSM}, considers all of the possible states that a managed software application can reach in its execution with respect to the QoS contract to satisfy (i.e., for both, the foreseen context situations to be faced by the managed application, and also for those unforeseen by the user).

Theorem 2 (Contract robustness with respect to context unpredictability)**.** *Let Q be a QoS contract according to Def. 4. $QoSC_{FSM}$ considers all of the possible states that a managed software application M, subject to Q, can reach in its execution.*

Proof. Let Z be the set of all possible states of execution that M can reach, as evaluated with respect to the QoS levels defined in Q. Then, for every $z \in Z$, z is considered by QoSC_{FSM}, given that

1. by Def. 4, Q defines a set of expected QoS levels, F, to be fulfilled by M; and
2. by Algorithm 1, a QoSC_{FSM} q is obtained from Q, such that its states, $q.STATES$, is initialized with the set F of fulfillment states, and the corresponding conditions of fulfillment in $q.\Psi$ (lines 2-16); and
3. by lines 18-19 of Algorithm 1, the Unstable and Exception states are added to $q.STATES$; and
4. by Def. 8 (the QoSC_{FSM} transition function), $\delta : \Sigma \times \Delta^U \rightarrow \{Fulfilled, Unstable, Exception\}$, that is, $\delta : \Sigma \times \Delta^U \rightarrow F \cup \{Unstable, Exception\}$, has 3 general conditions for each of its target transitions, depending on the context event (e) received from context monitors and the execution context (Δ) of M, which imply that the execution state of M, z, as evaluated with respect to the QoS-level conditions $q.\Psi$:
   (a) $z \in F$ is a fulfillment state given that it satisfies one of the QoS levels defined in F, that is, if $e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s))[match(r, \Re(s \,|\, e \in \eta(s))) \barwedge reconfig(\Re(s \,|\, e \in \eta(s)), \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F]$ (cf. condition (1), Sect. 5.1.3); or
   (b) $z = Unstable$ if $e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s))[match(r, \Re(s \,|\, e \in \eta(s))) \barwedge reconfig(\Re(s \,|\, e \in \eta(s)), \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = U]$ (cf. condition (2), Sect. 5.1.3); or
   (c) $z = Exception$ if $e \in \Sigma \barwedge \exists s : s \in STATES \,|\, \forall r : r \in \rho(s \,|\, e \in \eta(s))[\neg match(r, \Re(s \,|\, e \in \eta(s)))] \vee (e \in \Sigma^U \setminus \Sigma) \vee (e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in$

$\eta(s))[match(r, \Re(s \mid e \in \eta(s))) \overline{\wedge} \neg reconfig(\Re(s \mid e \in \eta(s)), \rho(s \mid e \in \eta(s)))])$ (cf. condition (3), Sect. 5.1.3).

In conclusion, any of the possible states, z, of M that can be reached in its execution ($z \in Z$) is either fulfilling one of the QoS levels specified in Q, or not fulfilling any of them, whichever this state z is. For this latter case, QoSC_{FSM} classifies these possible states as either Unstable or Exception. □

In other words, the expected states of fulfillment are the states corresponding to the QoS levels defined by the user in QoS contracts. These QoS levels are specified for each of the context conditions that the managed application can face in its execution, as foreseen by the user. However, given the unpredictable nature of context changes, it is easy to presume that the user can underestimate them (or omit them). Of course, this underestimation or omission originates the possibility for the managed application to reach unexpected and undesirable states. Nonetheless, even in these states the managed application is under controlled states (i.e., Exception or Unstable) that are automatically generated by our semantic model. In this way, QoS-CARE considers all of the possible states, with their respective transitions, that a managed software application can reach in its execution, with respect to its context (i.e., as specified in QoS contracts). Hence, we can conclude that the robustness conditions are realized in our QoS contract semantics, as part of our reconfiguration mechanism.

7. QoS-CARE Practical Feasibility

In contrast to other formal-based approaches, an important goal in the realization of our QoS contract semantics is an analysis of its practical feasibility, beyond analyzing its formal properties. This implies not only to show the approach applicability with a plausible example in the abstract (formulae) world, but building a *systematic* bridge between the abstract and actual running software world. Building this systematic bridge is not a trivial task. In fact, the key difference between the restricted applicability of a formal approach to a particular example and its generalized applicability to any software application demands the formulation of the approach as a sound and general semantics for the target artifacts of application, as presented in this paper.

To evaluate its applicability, we use QoS-CARE to preserve QoS contracts in three software applications with different sizes in a significant scale, and execute them in different configuration scenarios with the FraSCAti SCA runtime platform [46]. In particular, we analyze two aspects. First, the effect size produced by the size of the managed software applications in the reconfiguration effort required to be managed in QoS-CARE. Second, we measure its mean-time to reconfigure (MTTR) the managed applications by executing a benchmark of experiments on the configuration scenarios, as a way to evaluate QoS-CARE's settling-time.

The first software application corresponds to a complete version of the example used through this paper, that is, the mobile RVCS system subject to a QoS contract on *availability*. The second is a Web mashup application that dynamically composes and orchestrates the location service of Twitter[8] with a weather web service from different weather information providers, such as Google[9], Yahoo[10], and WebServiceX[11] to improve the service *readiness*. The third is an RMI

[8] https://dev.twitter.com/docs/api
[9] http://code.google.com/p/java-weather-api
[10] http://weather.yahooapis.com/forecastrss
[11] http://www.webservicex.net/ws/WSDetails.aspx?CATID=12&WSID=56

generic scalable sorting service subject to a QoS contract on computing-resources consumption.

For each of the configured scenarios, we present the effect size of the managed SCA software application in terms of QoS-CARE reconfiguration rules, and the experimental MTTRs measured from their execution. The complete description of the scenarios, including the application software with the corresponding QoS contracts and the analysis of the termination and consistency of the corresponding reconfiguration rules, are presented in [47].

7.1. Effect Size of Managed Application Sizes

The effect size of the managed applications sizes in the QoS-CARE effort required to reconfigure them are summarized in Table 3. The managed application sizes are expressed in terms of the application's number of SCA components, Java classes/interfaces, and physical lines of Java code (LOCs). Their effect size in QoS-CARE is expressed in terms of the number of reconfiguration rules required to satisfy their corresponding QoS contracts.[12]

Counting LOCs for Java programs is not a trivial task, in part because there is no agreement on the counting procedure apart of counting physical lines including or not blank lines and comments (e.g., compare the Eclipse Metrics[13] and the CodePro AnalytiX[14] projects). Additionally, in our case, the counting procedure has to deal also with SCA annotations in the Java code, and with the definition of SCA components written in XML, not mentioning that the FraSCAti middleware abstracts the implementation of crucial functionalities such as the intricacies of the invocation of remote procedures/methods for distributed components. Considering all these details, we count physical lines including comments and SCA annotations but not blank lines, for the Java files. We exclude the higher-level constructions and definitions of SCA components, but also the SCA middleware effects in the Java code, by not counting their lines but only the number of SCA components. Finally, to have an idea of the relationship between the number of SCA components and Java classes and interfaces, and also between Java classes and LOCs, we also counted the number of Java classes and interfaces.

The varying sizes of the managed SCA software applications in the table, compared to the number of reconfiguration rules required to satisfy their respective QoS contracts, show the applicability of QoS-CARE as independent of its managed application size. This applicability results from our semantic approach, which systematically processes the computational state of the managed application at runtime to build the graph model that QoS-CARE uses to transform and operate on. By virtue of the reconfiguration plan it synthesizes, the managed application is systematically and correspondingly transformed.

7.2. Reconfiguration Settling-time and Overhead

To measure the settling-time (MTTR) we used a set of benchmarks based on reconfiguration operations for each of the application scenarios, similar to the one described in Section 3. We used a context-events simulator to generate events notifying changes in each case, for instance changing the user's network access, from the intranet to the extranet, and vice versa, in the RVCS application. All of the experiments were performed on Intel i5@2.53Ghz processors with 4Gb of RAM running Mac OS X 10.6, using FraSCAti 1.4 with Java 1.6.0_23 allocated with 256MB of heap size.

[12] Their source code is available from http://gforge.icesi.edu.co/gf/project/seams

[13] http://metrics.sourceforge.net

[14] https://developers.google.com/java-dev-tools/download-codepro?hl=de-DE&csw=1

Table 3: Size of managed applications and its effect size in QoS-CARE reconfiguration rules

Managed Application	Application Element	Components	Classes and Interfaces	LOCs of Classes	Reconfig. Rules
RVCS	Monitoring Elements	1	1	84	6
	Application Logic	6	148	23,799	
	Total	7	149	23,883	
Web Mashup	Monitoring Elements	1	1	63	3
	Application Logic	4	24	1,512	
	Total	5	25	1,575	
Scalable Sorting	Monitoring Elements	1	1	71	2
	Application Logic	3	6	1,032	
	Total	4	7	1,103	

Table 4 presents the evaluation objective for each benchmark's configured scenario and the corresponding measured times. These objectives were selected considering the most representative operations on each case. Each scenario was executed 1,000 times.

Table 4: Settling-time (MTTR) Benchmark Scenarios and Results

	Evaluation Objective in Configured Scenario	Time (msec.)
RVCS	Local component deployment/undeployment[a]	49
	Local reconfiguration (total MTTR local)[a]	634
	"Remote" reconfiguration (total MTTR over loopback)[a]	640
	Remote reconfiguration (total MTTR over Internet)[b]	876
	QoS-CARE overhead (simulating 1 dummy event/3sec.)	3
Web Mashup	CBS_{TAG} from FraSCAti's SCA domain translation	15
	CBS_{TAG} transformation (one rule application)	14
	Reconfiguration plan transmission (473 bytes, 15 lines)	81
	Reconfiguration plan execution	47
	Total mean-time to reconfigure (MTTR)[c]	157
	QoS-CARE overhead (simulating 1 dummy event/3sec.)	3
Scalable Sorting	Remote component deployment[d]	719
	Remote reconfiguration (total MTTR over Internet)[d]	822
	QoS-CARE overhead (simulating 1 dummy event/3sec.)	3

[a] Client & Server in same machine.
[b] Instrumented through a REST service over a "common" Internet access (~1Mbps, 11 hops).
[c] Instrumented through a REST service over a loop-back connection.
[d] Instrumented through a REST service in a local-area network.

The obtained results confirm the applicability of QoS-CARE. On the one hand, the reconfiguration settling-times (MTTR) are small, between 157 and 876ms. On the other hand, compared to response times of regular Internet services, the overhead introduced by QoS-CARE, while running the managed applications in FraSCAti, is negligible (3ms).

8. Related Work

We state the contribution of our work in the *application* of formal models to the reliable and robust realization of QoS contract semantics, rather than in the formal models research area as such. By defining the QoS contracts' semantics we emphasize on the generality of this application of formal models at the level of actual software systems, beyond its application at the level of software abstractions. More precisely, our contribution is focused on satisfying QoS contracts under changing execution contexts, their reliable and robust fulfillment with respect to context uncertainty, and related properties of self-reconfiguring software systems. Thus, in this section we compare QoS-CARE with other approaches in these respects.

Regarding software contract management, several approaches have been proposed since the first ideas on functional contracts (as assertions or invariants) introduced by Floyd and Hoare [48, 49]. In the object-oriented paradigm, one of those approaches is the *design by contract theory* defined by Meyer [50]. Based on a characterization of the different conditions used in the so-called defensive programming, Meyer formulated systematic rules to guarantee routines to satisfy their contracts. Invariant violation, monitored at runtime, is automatically managed by the *rescue* clause, which must be handled by the programmer to restore a consistent state.

Conceptually, most of the approaches addressing QoS contracts follow the *rescue* clause idea of Eiffel. In contrast, in QoS-CARE the expected QoS-levels to fulfill can be seen, as a whole, as *context-dependent* "system invariants" to be satisfied continuously, according to context conditions. Nonetheless, our interpretation of *invariant* is not in the strict sense of Hoare (i.e., predicates expressed in static assertions that are true in the course of a specific sequence of instructions), but as predicates that are dynamically activated and deactivated at runtime according to changes in context conditions. A predicate must be activated and satisfied only when the respective context condition actually matches the current execution context.

Concerning robustness, Cheng *et al*. [51] and de Lemos *et al*. [34] identified context uncertainty as one of the most challenging problems faced by context-aware software systems. In this respect, the approach by Goldsby and Cheng [52] models dynamically adaptive systems as state-machines, with transitions as system reconfigurations, as QoS-CARE does. Inspired by the adaptability of living organisms, their approach models systems using UML diagrams that satisfy functional and non-functional invariants. Based on these diagrams, and by applying digital evolution techniques, they dynamically generate several target states for a given transition, and then assist the user to select the managed system with the most appropriate QoS provisions.

Other approaches that achieve self-adaptation triggered by context changes, although not addressing uncertainty, are the ones by Colombo *et al*. [39], Sykes *et al*. [53], and Cheng *et al*. [54]. The first uses reconfiguration rules in Event-Condition-Actions to model service (re)composition in BPEL. These rules associate BPEL workflows with reconfiguration policies used upon QoS level violation. The second exploits QoS information, provided by the user or monitored from the system, to make informed decisions when performing architectural adaptation. Its strategy generates all possible configurations and, in case of failure, switches to one of these configurations driven by this QoS information. The third performs architectural adaptation by using utility functions on repair strategies based on pre-computed configurations. The best configuration is selected and applied in response to system constraints violation. QoS-CARE differs from these three approaches in two aspects. First, it guarantees robustness by considering states of contract unfulfillment if the available reconfiguration options fail to fulfill a given QoS level in some relevant (but possibly unanticipated) context situation. Second, it guarantees other relevant reconfiguration properties such as termination, and consistency. That is, QoS-CARE does not

assume that the user-defined policies always cope with and manage unexpected context conditions as these approaches do. In actual software systems these are crucial situations that must be considered in their execution, and QoS-CARE effectively identifies and manages them with explicit states of contract unfulfillment.

Regarding the use of graph transformation systems, QoS-CARE shares similar principles as those managed by Bucchiarone *et al*. [44] and Ehrig *et al*. [45] to model dynamic software (re)configuration, and benefit from properties such as termination and deadlock-freeness. However, their approaches show these properties in particular application scenarios based on a given software abstraction with the purpose of guaranteeing self-repairing and self-healing properties when failures occur. Our purpose differs to theirs in that our use of graphs and state-machines as semantic denotations for QoS contracts not only generalizes the fulfillment of these contracts in different and actual running software systems, but also allows QoS-CARE to exploit *design-patterns at runtime* with the ultimate goal of preserving QoS levels. Another important difference is that QoS-CARE clearly separates running-software states from policy states, thus being able to have correct and valid software configuration states, even though not fulfilling the expected QoS levels.

Concerning QoS property preservation in component-based software, Léger *et al*. [15] address reliability for system reconfiguration in terms of structural consistency of components and connectors, while preserving system availability. They guarantee reliable reconfigurations in the Fractal platform [55] by extending its FScript textual language with transactional properties (Atomicity, (structural) Consistency, Isolation and Durability – ACID). Delaval *et al*. [16] proposed another approach, also based on Fractal, to guarantee safety. Inspired by control theory, their approach synthesizes static reconfiguration controllers. This synthesis is performed from a contract specification, similar to our translation of contracts to state machines. However, in these two approaches, the reconfiguration transition functions must be written by hand for each possible state defined in the state machine. Hnětynka and Plášil [56] preserve system structural properties in software reconfiguration. In order to prevent the dynamic reconfiguration from introducing architectural inconsistencies, their approach characterize three reconfiguration patterns that specifically aim at avoiding the degradation of the system structure. Then, they restrict the set of reconfigurations to be used to only those conforming to these patterns. In contrast to QoS-CARE, in this approach patterns are used *a priori*, before reconfigurations, to preserve structural conformance. This property, nonetheless, is also preserved by QoS-CARE *a posteriori*, by verifying it after each reconfiguration.

In the Web services area, El Hadad *et al*. [57] propose a design-time selection approach for composing Web services automatically, where transactional and QoS requirements are integrated in the selection process. Their approach combines a transactional-aware service selection with a QoS-aware service selection, using local QoS optimization to reduce the set of possible transactional solutions. The purpose of this combination is to satisfy the global transaction requirements of workflow-based systems. The workflow activities are fulfilled with Web services, such that the combination of services satisfy also the systems' QoS requirements. Even though this proposal and QoS-CARE have in common the goal of preserving QoS properties, they differ in several aspects. First, this proposal manages services, that is, software artifacts as black boxes whose internal structure is not important, whereas QoS-CARE manages sets of components and their connectors (i.e., interconnected software structures providing functionalities). Second, the services selected by the referred approach are composed based on the structure of a given workflow specified by the user, and based on this workflow, the system's QoS properties are evaluated to find an optimal combination. In contrast, in QoS-CARE the user specifies design-patterns as sets

of components and corresponding connectors, which determine particular levels of QoS properties. These design patterns are matched against (and replaced in) the software system structure, whose structure is not predefined. Finally, the referred proposal targets software systems with a static structure, that is, whose structure does not change at runtime (i.e., the workflow given by the user is immutable at runtime). QoS-CARE targets dynamically self-reconfigurable software systems, and to support this reconfiguration capability at runtime it provides in its architecture the components to monitor, analyze, plan, and execute the structural modifications in the managed system that are required to fulfill a particular level of a QoS property.

From the formal semantics point of view, Fiadeiro and Lopes [58] present a formalization for the dynamic reconfiguration of business process workflows in terms of service discovery, binding, and orchestration operations for the SOA domain. They propose several levels of abstraction, using different formalisms (linear-time logic, graphs, and partial algebras) at each level, to model the structural and behavioural corresponding aspects. The formalization is then proposed as a semantic domain with a corresponding operational semantics that enables an architecture-description language (ADL) to be extended with dynamic reconfiguration operations. Braga *et al*. [7] formalize the semantics of a QoS contract language using operational calculus rules driven by a state machine. To guarantee the satisfaction of QoS constraints on the resulting states, they translate QoS specifications to the Maude model checking tool. As in QoS-CARE, this semantics considers actual monitored context conditions in its transitions.

QoS-CARE shares with these latter approaches the formalization of QoS properties and contract semantics, but differs from them by guaranteeing robustness. Moreover, our characterization of *unstable* states constitutes a strategy for detecting (and acting upon) the problem of reaching stability, as explained in previous sections. We evidence this problem in oscillations failing to fulfill a given QoS level from our runtime-monitored context evaluation in reconfiguration transitions. Nonetheless, the stability problem remains a major challenge for controlling software self-adaptation still asking for a general solution, as identified by Hellerstein *et al*. [59].

9. Conclusion

In this paper we have presented the definition, implementation, and evaluation of a two-layered denotational semantics for QoS contract preservation in component-based software systems under changing context situations, which guarantees reliability and robustness.

In the governing layer of this semantics we interpret a QoS contract as an extended FSM, with QoS-level conditions as states, and reconfiguration rules as transitions. The extended FSM guarantees contract robustness with respect to context unpredictability by characterizing the transitions and states of contract fulfillment and unfulfillment. In the operating layer, FSM states are mapped to software structures denoted as TAGs, and transitions to software re-configurations through extended TAG transformation systems. The extended transformation system makes it possible to exploit *design patterns* at runtime, despite they have been applied until now at design-time to determine quality attributes. We implemented this contract semantics in QoS-CARE, a framework for developing context-aware and self-adaptive mechanisms, that is able to execute and reconfigure component-based software applications for fulfilling its associated QoS contracts in an autonomous, reliable and robust way. Thus, given a QoS contract, QoS-CARE process it as the high-level policy that governs the behavior of its reconfiguration mechanism over the managed software to maintain fulfilled its expected QoS levels.

The relevance of our two-layered denotational semantics for QoS contracts is threefold. First, it generalizes the applicability of QoS-CARE by providing denotations for QoS contract ele-

ments and software structures, systematically. Second, it bridges the formal (abstract) and actual running-software worlds with a function that translates the SCA structure of a software in execution to its operating denotation. Third, given a QoS contract, it guarantees atomicity and robustness, and verifies the reconfiguration termination and component-based structural-conformance conditions. Additionally, our evaluation results of the actual implementation of our formal semantics in the QoS-CARE framework, as applied to three application scenarios, show its practical feasibility, usability, and applicability.

As worth of future work, we consider three important aspects of our contribution. First, the formalism for specifying reconfiguration rules in QoS-CARE requires adequate training, despite the current graph-tools support. In order to make QoS-CARE more usable by non-experts we are developing a more legible and usable concrete syntax to specify these rules. The idea is to define a domain-specific language (DSL) with automated tools to assist the user in the writing of these rules in a more familiar notation, such as the used for component-based applications in SCA. Second, on leveraging design patterns at runtime, not only to address QoS contracts as we showed with QoS-CARE, but also with other purposes. For instance, design-patterns at runtime could help to develop behaviour models to predict stability and efficient resource use in self-adaptive software systems. Finally, on improving contract robustness to context unpredictability. The states of contract *unfulfillment* managed by QoS-CARE constitutes only one step towards understanding how to achieve general robustness to context uncertainty and related problems of self-adaptation. For instance, as mentioned before, our unfulfillment-unstable state address the stability problem by detecting unstability, although not preventing it.

Acknowledgments. This work was funded in part by the Ministry of Higher Education and Research of Nord-Pas de Calais Regional Council and FEDER, CPER 2007-2013, and during the tenure of an "*Alain Bensoussan*" ERCIM Fellowship by the third author.

References

[1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, D. Watkins, Making components contract aware, IEEE Computer 32 (1999) 38–45.

[2] V. X. Tran, H. Tsuji, A Survey and Analysis on Semantics in QoS for Web Services, Intl. Conf. on Advanced Information Networking and Apps. (2009) 379–385.

[3] S. Frølund, J. Koistinen, Quality of services specification in distributed object systems design, in: Procs. of 4th Conf. on Object-Oriented Technologies and Systems, volume 4, USENIX Association, 1998, pp. 179–202.

[4] A. Keller, H. Ludwig, The wsla framework: Specifying and monitoring service level agreements for web services, J. Netw. Syst. Manage. 11 (2003) 57–81.

[5] S. Röttger, S. Zschaler, CQML+: Enhancements to CQML, in: Procs. of 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Cépaduès-Éditions, 2003, pp. 43–56.

[6] S. Becker, Quality of Service Modeling Language, in: I. Eusgeld, F. Freiling, R. Reussner (Eds.), Dependability Metrics, volume 4909 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2008, pp. 43–47.

[7] C. Braga, F. Chalub, A. Sztajnberg, A formal semantics for a quality of service contract language, Electronic Notes of Theoretical Computer Science 203 (2009) 103–120.

[8] A. Cansado, C. Canal, G. Salaün, J. Cubo, A formal framework for structural reconfiguration of components under behavioural adaptation, Procs. of the 6th Intl. Workshop FACS 2009. ENTCS 263 (2010) 95 – 110.

[9] P. Collet, R. Rousseau, T. Coupaye, N. Rivierre, A Contracting System for Hierarchical Components, in: Procs. of 8th Intl. Symp. of Component-Based Software Engineering, volume 3489 of *LNCS*, Springer, 2005, pp. 187–202.

[10] H. Chang, P. Collet, Patterns for integrating and exploiting some non-functional properties in hierarchical software components, in: Procs. of 14th IEEE Intl. Conf. and Workshops on the ECBS'07, IEEE CS, 2007, pp. 83–92.

[11] J. Y. Lee, J. W. Lee, D. W. Cheun, S. D. Kim, A Quality Model for Evaluating Software-as-a-Service in Cloud Computing, in: Procs of 7th Intl. Conf. on Software Engineering Research and Apps., IEEE CS, 2009, pp. 261–266.

[12] M. Comuzzi, B. Pernici, A framework for QoS-based Web service contracting, ACM Transactions on the Web 3 (2009) 10:1–10:52.

[13] C. Szyperski, Component Software: Beyond Object-Oriented Programming, ACM Press/Addison-Wesley, 1998.
[14] G. T. Heineman, W. T. Councill (Eds.), Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Longman, 2001.
[15] M. Léger, T. Ledoux, T. Coupaye, Reliable Dynamic Reconfigurations in a Reflective Component Model, in: Procs. of 13th Intl. Symp. of Component-Based Software Eng., volume 6092 of *LNCS*, Springer, 2010, pp. 74–92.
[16] G. Delaval, É. Rutten, Reactive Model-Based Control of Reconfiguration in the Fractal Component-Based Model, in: Procs. of 13th Intl. Symp. Component-Based Software Engineering, volume 6092 of *LNCS*, pp. 93–112.
[17] L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, QoS-Aware Middleware for Web Services Composition, IEEE Transactions on Software Engineeering 30 (2004) 311–327.
[18] G. Tamura, R. Casallas, A. Cleve, L. Duchien, QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs, in: Formal Aspects of Component Software, volume 6921 of *LNCS*, Springer, 2011, pp. 34–52.
[19] N. Villegas, H. Muller, G. Tamura, L. Duchien, R. Casallas, A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems, in: Procs. of 6th Intl. Symp. on Software Engineering for Adaptive and Self-Managing Systems, ACM, 2011, pp. 80–89.
[20] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, Upper Saddle River, NJ, 1996.
[21] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, Reading, Mass, second edition, 2003.
[22] P. Kruchten, H. Obbink, J. Stafford, The Past, Present, and Future for Software Architecture, IEEE Software 23 (2006) 22–30.
[23] F. Buschmann, K. Henney, D. Schmidt, Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series), John Wiley & Sons, 2007.
[24] P. Clements, M. Shaw, The Golden Age of Software Architecture Revisited, IEEE Softw. 26 (2009) 70–72.
[25] C. A. Gunter, Semantics of Programming Languages: Structures and Techniques, MIT Press, Cambridge, MA, USA, 1992.
[26] M. Barbacci, M. H. Klein, T. A. Longstaff, C. B. Weinstock, Quality Attributes, Technical Report CMU/SEI-95-TR-021, CMU/SEI, 1995.
[27] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[28] J. Ramachandran, Designing Security Architecture Solutions, John Wiley & Sons, Inc., New York, NY, USA, 2002.
[29] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, K. Togashi, Secure Design Patterns, Technical Report CMU/SEI-2009-TR-010, CMU/SEI, CERT Program, 2009.
[30] M. Kircher, P. Jain, Pattern-Oriented Software Architecture: Patterns for Resource Management, John Wiley & Sons, 2004.
[31] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Wesley Professional, first edition, 2004.
[32] W. Zeng, X. Zhuang, J. Lan, Network friendly media security: Rationales, solutions, and open issues, in: Procs. of the 2004 Intl. Conf. on Image Processing (ICIP), IEEE, 2004, pp. 565–568.
[33] S. Krakowiak, Middleware architecture with patterns and frameworks, 2009.
[34] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, in: R. de Lemos, H. Giese, H. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems 2, volume 7475 of *LNCS*, Springer, 2013, pp. 1–32.
[35] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: Procs. of Applications of Graph Transformation with Industrial Relevance, volume 3062 of *LNCS*, Springer-Verlag, 2004, pp. 446–453.
[36] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer-Verlag, 2009.
[37] J. O. Kephart, D. M. Chess, The Vision of Autonomic Computing, IEEE Computer 36 (2003) 41–50.
[38] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2006.
[39] M. Colombo, E. Di Nitto, M. Mauri, SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules, in: Intl. Conf. on Service-Oriented Computing, volume 4294 of *LNCS*, 2006, pp. 191–202.
[40] A. V. Aho, J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.
[41] D. Harel, On Visual Formalisms, Communications of the ACM 31 (1988) 514–530.

[42] R. Heckel, J. M. Küster, G. Taentzer, Confluence of typed attributed graph transformation systems, in: Proceedings of the First International Conference on Graph Transformation, ICGT '02, Springer-Verlag, 2002, pp. 161–176.
[43] H. Ehrig, U. Prange, G. Taentzer, Fundamental theory for typed attributed graph transformation, in: Proc. of ICGT'04, volume 3256, pp. 161–177.
[44] A. Bucchiarone, P. Pelliccione, C. Vattani, O. Runge, Self-repairing systems modeling and verification using agg, in: IEEE/IFIP WICSA/ECSA, IEEE, 2009, pp. 181–190.
[45] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, P. Pelliccione, Formal analysis and verification of self-healing systems, in: FASE'10, volume 6013 of *LNCS*, Springer, 2010, pp. 139–153.
[46] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, J.-B. Stefani, Reconfigurable SCA Applications with the FraSCAti Platform, in: Procs. of 6th Intl. Conf. on Services Computing, SCC'09, IEEE, 2009, pp. 268–275.
[47] G. Tamura, QoS-CARE: A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration, Phd thesis, University of Lille 1 - Science and Technology and University of Los Andes, 2012.
[48] R. Floyd, Assigning Meaning to Programs, in: Proc. of Symposium on Applied Mathematics, volume 19, A.M.S., 1967, pp. 19–32.
[49] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580.
[50] B. Meyer, Applying "Design by Contract", Computer 25 (1992) 40–51.
[51] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle, Software Engineering for Self-Adaptive Systems: A Research Roadmap, in: Software Engineering for Self-Adaptive Systems, LNCS, Springer-Verlag, 2009, pp. 1–26.
[52] H. J. Goldsby, B. H. Cheng, Automatically generating behavioral models of adaptive systems to address uncertainty, in: Procs. of 11th Intl. Conf. on Model Driven Engineering Languages and Systems, Springer, 2008, pp. 568–583.
[53] D. Sykes, W. Heaven, J. Magee, J. Kramer, Exploiting non-functional preferences in architectural adaptation for self-managed systems, in: Procs. of 2010 ACM Symposium on Applied Computing, ACM, 2010, pp. 431–438.
[54] S.-W. Cheng, D. Garlan, B. Schmerl, Architecture-based self-adaptation in the presence of multiple objectives, in: Procs. of 2006 Intl. Workshop on Self-Adaptation and Self-Managing Systems, ACM, 2006, pp. 2–8.
[55] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The Fractal Component Model and its Support in Java, Software Practice & Exper. 36 (2006) 1257–1284.
[56] P. Hnětynka, F. Plášil, Dynamic reconfiguration and access to services in hierarchical component models, in: Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063, Springer-Verlag, 2006, pp. 352–359.
[57] J. El Hadad, M. Manouvrier, M. Rukoz, TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition, IEEE Transactions on Services Computing 3 (2010) 73–85.
[58] J. L. Fiadeiro, A. Lopes, A model for dynamic reconfiguration in service-oriented architectures, in: Procs. of 4th European Conference on Software Architecture, Springer-Verlag, 2010, pp. 70–85.
[59] J. L. Hellerstein, S. Singhal, Q. Wang, Research Challenges in Control Engineering of Computing Systems, IEEE Transactions on Network and Service Management 6 (2009) 206–211.