QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs

Gabriel Tamura, Rubby Casallas, Anthony Cleve, Laurence Duchien

QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs

Gabriel Tamura[1,2], Rubby Casallas[1], Anthony Cleve[2], and Laurence Duchien[2]

[1] University of Los Andes, TICSw Group, Cra. 1 N° 18A-10, Bogotá, Colombia
[2] INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, University of Lille 1, France
gabriel.tamura@inria.fr,rcasalla@uniandes.edu.co,anthony.cleve@inria.fr,
laurence.duchien@inria.fr

Abstract. In this paper we focus on the formalization of component-based architecture self-reconfiguration as an action associated to quality-of-service (QoS) contracts violation. With this, we aim to develop on the vision of the component-based software engineering (CBSE) as a generator of software artifacts responsible for QoS contracts. This formalization, together with a definition of a QoS contract, forms the basis of the framework we propose to enable a system to preserve its QoS contracts. Our approach is built on a theory of extended graph (e-graph) rewriting as a formalism to represent QoS contracts, component-based architectural structures and architecture reconfiguration. We use a rule-based strategy for the extensible part of our framework. The reconfiguration rules are expressed as e-graph rewriting rules whose left and right hand sides can be used to encode design patterns for addressing QoS properties. These rules, given by a QoS property domain expert, are checked as *safe*, i.e., terminating and confluent, before its application by graph pattern-matching over the runtime representation of the system.

1 Introduction

In the last ten years, Component-based Software Engineering (CBSE) has evolved based on a fundamental vision of the components as a contract or obligations-responsible software artifacts [1]. On this vision, CBSE has been used as a fundamental approach for engineering software systems in a wide variety of forms. These forms include the building of systems from contract-compliant components to abstracting reflection mechanisms at the component-level (i.e., composite, component, port, connection) to support self-adaptive systems. Even though a lot of research has been conducted on how to make components guarantee contracts on individual functionality, making component-based systems to be QoS contracts-aware is another important part of the same research question: this kind of contracts constitute the base to differentiate and negotiate the quality of the service or provided functionality at the user level.

Nonetheless, providing a component-based software system with reconfiguration capabilities to preserve its QoS contracts presents several difficulties: (i) the expression of the QoS contract itself, given that it must specify the different contextual conditions on the contracted QoS property, and the corresponding guaranteeing actions to be performed in case of the QoS contract disruption [13,15]; (ii) in contraposition to functional contracts, which can be checked statically, QoS contracts are affected by global and extra-functional behaviour that must be evaluated at runtime. This evaluation requires also dynamic monitoring schemes, different to the static ones usually found in current systems [7]; (iii) several reconfiguration strategies can be used to address each desirable condition on a QoS property. These strategies are provided by different disciplines (e.g., those related to performance, reliability, availability and security), and constitute a rich knowledge base to be exploited. Nonetheless, due to their diversity of presentation in syntax and semantics, it is difficult to manage them uniformly, thus existing approaches use them as fixed subsets [2]; (iv) the reconfiguration process is required to guarantee both, the preservation of the system integrity as a component-based software system, and the correct and safe application of the reconfiguration strategy. This requirement is specially challenging if the strategies are parametrized, for instance, by using rules, still being a research issue in self-reconfiguring approaches [12].

On the treatment of software contracts several works have been proposed. Notably among them, the *design by contract* specification of the Eiffel programming language [16] and the *Web Service Level Agreement (WSLA)* initiative [15]. The design by contract theory, one of the most inspiring in the object-oriented programming paradigm, makes routines self-monitoring at compile-time by using assertions as integral parts of the source code to be checked at runtime. The violation of an assertion, such as a class invariant, is automatically managed by standard mechanisms like the *rescue* clause. The programmer must handle it appropriately to restore a consistent state. This idea was later generalized by Beugnard et al. to four types of software contracts, including those based on QoS, though not fully developed [3]. On the other side, WSLA specifies QoS contracts independent from the source code, thus involving conditions based on the actual context of execution. The WSLA includes a guaranteeing action in response to disrupted SLAs, but the semantics of this action is limited to operations such as event notification [15]. However, despite these and other many advances, the development of a well-founded theory to manage QoS contracts in component-based systems is still a challenging question.

Our goal in this paper is to formally model the *architecture reconfiguration* of a component-based (CB) system as an action performed by itself. These actions are performed in response to the disruption of QoS contracts, in the spirit of the Eiffel's *rescue* clause in object-oriented programming. By doing this, we aim to develop on the vision of the CBSE as a sound base to produce software systems enabled to automatically and safely reconfigure themselves by reconfiguring their abstract (reflection) architectures at runtime. For such structural reconfigurations, a system architect may reuse design patterns from other disciplines with the purpose of restoring QoS contracts, thus preserving them.

Our approach is built on the theory of extended graph (e-graph) rewriting proposed in [10], as a formalism to represent QoS contracts, component-based architectural structures and architecture reconfiguration. For the self-reconfiguration, we use a parametrized, *rule-based* strategy. That is, the reconfiguration possibilities are expressed as e-graph rewriting rules whose left and right hand sides can be used to encode variations of design patterns for addressing QoS properties. These rules are applied by graph pattern-matching over the system runtime e-graph representation when it is notified with events related to the violation of the corresponding properties.

The contribution of this paper is twofold. We provide (i) formal definitions for QoS contracts, CB system reflection and reconfiguration rules, in a unified framework (i.e., syntax and semantics). This allows the verification of CB structural rules of formation to be checked; and (ii) a well-founded basis for a system to manage its own reconfigurations to address the disruption of its associated QoS contracts. Once parametrized with a specific set of rules, the system can be checked as terminating (the process of rule application is guaranteed to end) and confluent (the rule application order is irrelevant and always produce the same result).

This paper is organized as follows. Section 2 presents our motivation and proposal scope. Section 3 introduces a reliable video-conference system as an example scenario to illustrate our proposal. Section 4 presents our formalization for QoS contracts-ware system reconfiguration by using e-graphs. Section 5 analyze the properties of our reconfiguration system as a result of its formalization. Section 6 compares our approach with similar proposals. Finally, Section 7 concludes the paper and anticipates future work.

2 Motivation and Scope

As defined by Oreizy et al., self-adaptive software evaluates its own behaviour at runtime and modifies itself whenever it can determine that it is not satisfying its requirements [17,20]. In their proposal, they defined the system adaptation as a cycle of four phases: (i) monitoring of context changes; (ii) analysis of these changes to decide the adaptation; (iii) planning responsive modifications over the running system; and (iv) deploying the modifications. In our proposal, we focus on the planning phase considering self-reconfiguration at the component level, triggered by sensible changes in context that affect the fulfillment of contractual QoS properties. Other component-based proposals such as COSMOS [9] and MUSIC [18] can be used for more general functionalities of context monitoring and analysis phases, meanwhile those like Fractal [4] and OSGi [21] for the component management at the deployment and execution phases.

Our motivation in this paper is to define a safe, rule-based framework to address QoS contracts violation in CB systems through the reconfiguration of the components-architecture, meaning: (i) (*rule-based reconfiguration*) the addition or removal of software components and connectors at runtime, as specified by parametrized rules given by a QoS property domain expert or a software QoS architect; (ii) (*safe-1*) these rules can be checked to be *terminating* and *confluent*, i.e., their application can be guaranteed to finish the production of the reconfiguration actions in a deterministically way. This verification is done despite the rules given being whether or not pertinent for the QoS property preservation, but correct in their definition; (iii) (*safe-2*) the QoS property domain expert is concerned only with rule specification, not with the specific procedure to apply it; (iv) (*safe-3*) once executed the reconfiguration actions into the runtime system, its CB-structural conformance can be verified.

3 Running Example

We illustrate the requirements for dynamic reconfiguration with a simplified version of a reliable mobile video-conference system (RVCS). To the user, the service is provided through a video-conference client subject to a QoS contract on its reliability. Thus, software clients are expected to be responsible for maintaining the service to the user in a "smart" way, as illustrated in Fig. 1. Note that addressing these requirements statically (e.g., with *if-then* clauses on context conditions) would not be satisfactory: as the video-conference requires bi-directionality, this would introduce synchronization issues between the client's and server's conditions, being their respective contexts not necessarily the same.

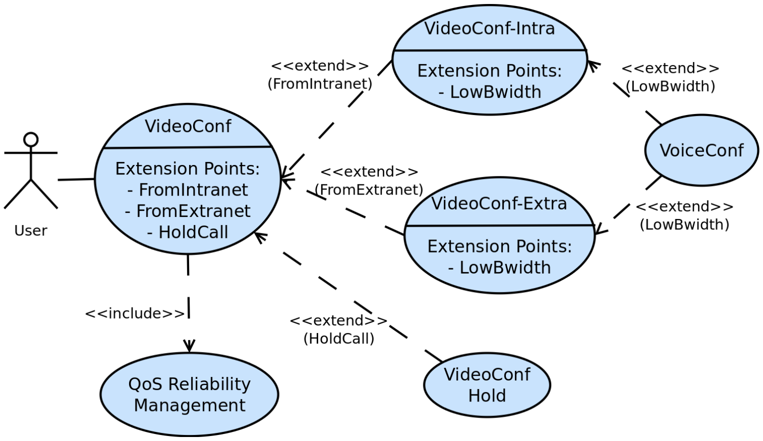

Fig. 1. Use case diagram for the requirements of the RVCS example. Connections from the intranet are considered secure, thus clear communication channels can be used. From the extranet, confidential channels are required to be configured. In case of no connection, the call must be put on hold. If the user goes into a low-bandwidth area, the system must reconfigure itself to drop the bi-directional video signals. The *QoS Reliability Management* has the responsibility of reconfiguring the system architecture to address the QoS contract violation in each case (taking into account the system's actual state) in a transparent way.

In this example, reliability is interpreted following [2], i.e., to ensure the continued availability of the video-conferencing service hosted by a corporate network. The corporate network requires all clients to access the intranet through connections guaranteeing confidentiality. Thus, even though the contract is on the QoS property of reliability, it involves two sub-properties, confidentiality and availability. On these two sub-properties, the contractual interest is on establishing the minimum levels for service acceptability (*service level objectives*), under the possible contextual conditions of system execution (cf. Tables 1 and 2).

Table 1. QoS contractual conditions and corresponding service level objectives for the confidentiality property (based on access to corporate network).

Contextual Condition	Service Level Objective
CC1: Connection from Intranet	Clear Channel
CC2: Connection from Extranet	Confidential Channel
CC3: No Network Connection	Call on Hold

Initially, assume the mobile user joins a video conference from her office at the corporate building, e.g., from an intranet WiFi access-point. In this state, as the contractual condition $CC1$ in Table 1 requires a clear-channel communications configuration, the system is expected to configure itself to satisfy that condition. A second system state is reached when she moves from her office to outside of the company building thus connecting through any of the available extranet wireless access-points, such as GSM or UMTS. This context change, signaled by a new contextual condition, disrupts the confidentiality contract that was being fulfilled by the actual system configuration. In this new state, according to condition $CC2$, a confidential-channel configuration on the mobile is required. The expected system behaviour is then to reconfigure itself in response to this change, in a transparent way, adopting, for instance, one of the strategies for secure multimedia transport like those defined in [23,19], thus restoring the contract. The corresponding contrary reconfiguration would apply whenever she moves back to an access-point covered by the intranet. If there are several available network access-points, a cost function should be used to choose the cheapest. Finally, whenever there is no network connection by any access-point, the call must be put on hold awaiting for automatic reconnection, just expressing that this is preferable to the alternative of dropping the service.

For illustration purposes, Table 2 establishes the minimum expected service, according to the network bandwidth, independent of the network access-point location.

4 E-Graph Modeling of QoS Contracts-Based System Reconfiguration

Given that QoS properties are dependent on system architecture, we build our proposal for making CBSE systems to be QoS contracts-responsible on a formal

Table 2. QoS contractual conditions and corresponding service level objectives for the availability property (based on network bandwidth in kbit/s).

Contextual Condition	Service Level Objective
CC4: $BandWidth \leq 12$	Call on Hold
CC5: $12 < BandWidth \leq 128$	Voice Call
CC6: $128 < BandWidth$	Voice and Video Call

modeling for component-based architecture self-reconfiguration. This formalization is built on the extended theory of graph transformation given in [10].

For a CBSE system to be QoS contracts-responsible in an autonomous way, it requires (i) to have a structural representation of itself at the component level (i.e., to be reflective) [8]; (ii) to have a representation of its QoS contracts: the service level objectives for each of the contractual QoS properties, under the different contextual conditions; (iii) to be self-monitoring, that is, to identify and notify events on the contractual QoS properties violation; and (iv) to apply the architecture reconfiguration to restore the violated QoS property condition, as specified in the QoS contracts.

In Sect. 4.1 we recall the base definitions of e-graphs given in [10]; then, we use these definitions in sections 4.2 and 4.3 as a unified formalism to represent reflection structures for component-based systems and QoS contracts respectively. Finally, in Sect. 4.4 we present our proposal for architecture reconfiguration based on e-graph rewriting rules, illustrating how these defined constructs give support for reflective, autonomous and QoS contracts-based self-reconfiguring systems.

4.1 Extended Graphs: Base Definitions

Definition 1 (E-Graph). *An E-Graph is a 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);*
- $source_1 : E_1 \rightarrow V_1$; $source_2 : E_2 \rightarrow V_1$; $source_3 : E_3 \rightarrow E_1$ *are the source functions for the edges; and*
- $target_1 : E_1 \rightarrow V_1$; $target_2 : E_2 \rightarrow V_2$; $target_3 : E_3 \rightarrow V_2$ *are the target functions for the edges, as depicted in Fig. 2.*

Definition 2 (E-Graph morphism). *An e-graph morphism* f *between e-graphs* G *and* H, $f : G \rightarrow H$, *is a tuple* $(f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2}, f_{E_3})$ *where* $f_{V_i} : G_{V_i} \rightarrow H_{V_i}$ *and* $f_{E_j} : G_{E_j} \rightarrow H_{E_j}$ *for* $i = 1, 2,\ j = 1, 2, 3$*, such that* f *commutes with all source and target functions*[2] *(cf. Fig. 3).*

[2] Note that E-Graphs combined with E-Graph morphisms form the category *EGraphs*. See [10] for more details on this topic.

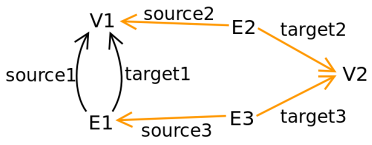

Fig. 2. E-Graph definition. An e-graph extends the usual definition of a base graph, $(V_1, E_1, source_1, target_1)$, with (i) V_2, the set of attribution nodes; (ii) E_2 and E_3, the sets of attribution edges; and (iii) the corresponding *source* and *target* functions for E_2 and E_3, used to associate the attributes for V_1 and E_1, respectively, to V_2.

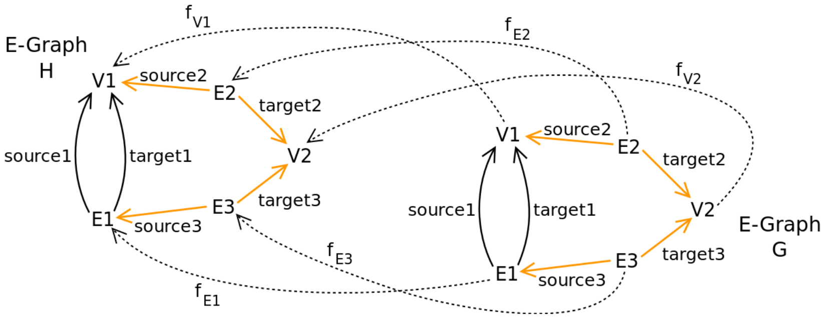

Fig. 3. E-Graph morphism illustration example f between e-graphs G and H, $f : G \rightarrow H$. E-graph morphisms are used as typing relationships between e-graphs.

4.2 System Reflection

For a system to self-reconfigure at runtime, it is required to be reflective. That is, it must be able to identify and keep track of the individual elements that are to be involved in reconfiguration operations [8]. In our case, the reflection structure is defined on a component-based structure that comprises the CBSE component, port, port type and connector elements. Composites are abstracted as components, as we address structural reconfiguration at the system level.

Definition 3 (Component-Based Structure - CBS). *The component-based structure, CBS, is the tuple $(G, DSig)$, where*

- *$DSig$ is a data signature over the disjoint union $String + PortRole$ and $PortRole = \{Provided, Required\}$, with the usual CBSE interpretations;*
- *G is the e-graph $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that $V_1 = \{SReflection, Component, Port, PortType, Connector\}$; each of the data nodes is named after its corresponding sort in $DSig$, $V_2 = String{+}PortRole$; $E_1 = \{component, port, provided, required, type\}$, $E_2 = \{cname, pname, ptype, role, c.QoSProvision, p.QoSProvision, ct.QoSProvision\}$, $E_3 = \{\}$; and the functions $(source_i, target_i)_{i=1,2,3}$ are defined as depicted in Fig. 4.*

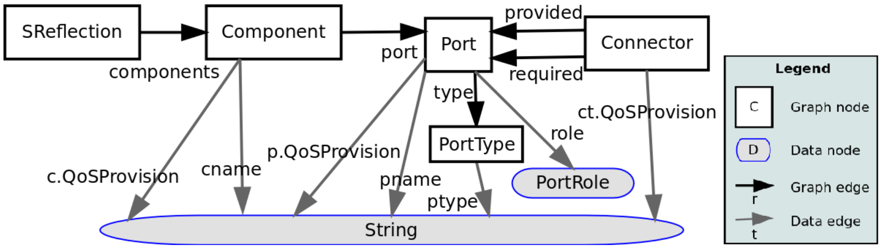

Fig. 4. The Component-Based Structure, CBS, is defined as an e-graph where each of the graph nodes represents each of the CBSE elements. The graph edges correspond to the relationships among these elements, meanwhile the data edges, to their corresponding attributes; *QoSProvision* is a special attribute for components, ports and connectors to express that they warrant a particular QoS condition, such as providing a secure connection to a network. The data nodes represent the types of these attributes.

Definition 4 (Component-Based System Reflection). *Given S the computational state of a running component-based system, its corresponding reflection state, R_S, is defined as $R_S = (G, f_S, t)$, where G is the e-graph that represents S through the one-to-one function $f_S : S \to G$, and t is an e-graph morphism $t : G \to CBS$.*

That is, S represents the state of each of the system components, ports and connectors as maintained in a component platform such as FRACTAL or OSGi. The feasibility of f_S results from Def. 3 (CBS) and the e-graph morphism t. $R_S.Component$ denotes the set of components in R_S, i.e., $R_S.Component = \{c | c \in G_{V1} \wedge t_{V1}(c) = Component\}$ (analogously for the other CBS elements). The purpose of f_S is to map the system architecture into the e-graphs domain, in which the architecture reconfiguration is operated. Once reconfigured, we use f_S^{-1} to perform the reconfiguration back in the actual runtime component-based system.

Example 1 (Video Conference System). Figures 5 and 6 illustrate, respectively, the runtime component-based system structure of our video-conference example and its corresponding system reflection state, when configured to be connected from the intranet (i.e., with a clear-channel connection).

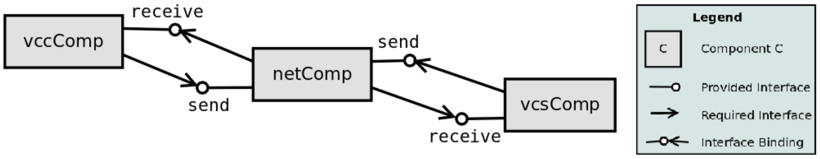

Fig. 5. Runtime system structure for Ex. 1 with a clear channel connection.

The components of Fig. 5 are represented in Fig. 6 as exactly the video conference client with its network connection ($vccComp$ and $netComp$) and the server ($vcsComp$). The other elements in represent their ports ($vccP1$, $vccP2$ and so on), and these port's actual connections ($con1$, $con2$ and so on).

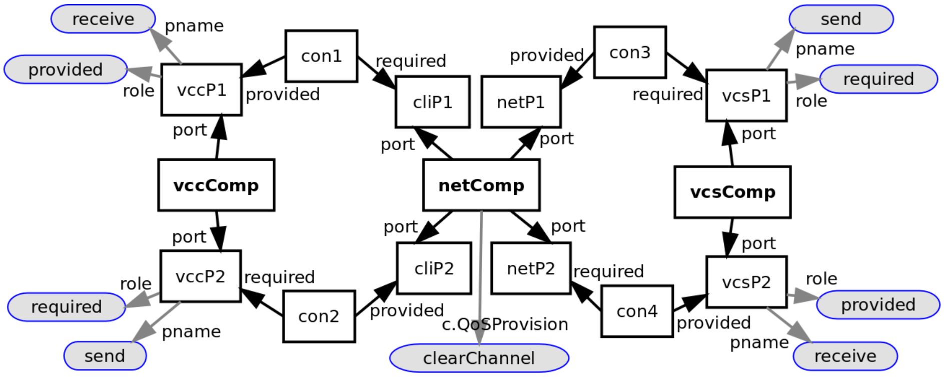

Fig. 6. Runtime system reflection structure, in e-graph notation, for the runtime system of Fig. 5 (i.e., when connected from the intranet). The $netComp$ component, providing a network connection, is responsible for maintaining a $clearChannel$ connection, as expressed by its $c.QoSProvision$ attribute. Further details omitted for space.

4.3 QoS Contracts

A *QoS contract* is a specification of the guarantees on QoS properties under specific conditions for a given functionality, as offered by a system or service provider to *any* of its potential clients [14,3]. In this sense, a QoS contract is an invariant that a system must preserve, 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 property condition must be monitored and the system must act upon its violation in order to have the possibility of restoring it opportunely.

For a system to address its QoS contracts' violation, it must incorporate and manage these contracts internally. Given our formal modeling of a component-based system as a realization of system reflection, we use the same formal framework to define QoS contracts as a manageable part of the system.

Definition 5 (QoS Contract). *Given $QoSDSig$ the usual data signature over the disjoint union $String + Boolean$, a QoS contract is a tuple (C, ct), where*

– *C is an e-graph representing the contract instance;*

– *ct is an e-graph morphism $ct : C \rightarrow Q$, where Q is the e-graph reference definition for QoS contracts, $(V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that $V_1 = \{QoSContract, QoSProperty, QoSMonitor, QoSGuarantor, SLOObligation, QoSRule\}$; each of the data nodes is named after its corresponding sort in QoSDSig, $V_2 = String + Boolean$; $E_1 = \{property, obligation, monitor, guarantor, ruleSet\}, E_2 = \{gname, pname, mname, rname, SLOPredicate, contextCondition, isActive\}, E_3 = \{\}$; and the functions $(source_i, target_i)_{i=1,2,3}$ are defined as depicted in Fig. 7.*

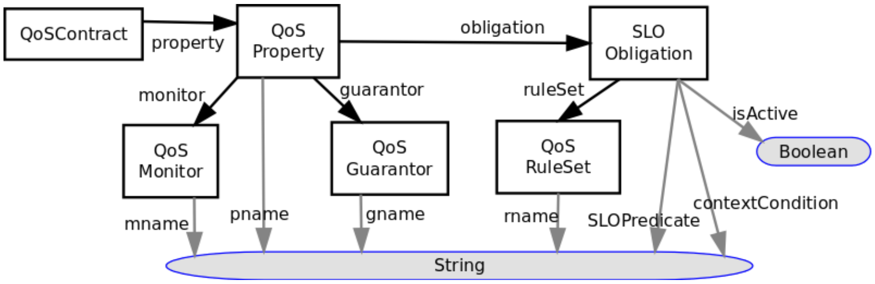

Fig. 7. E-graph reference definition for QoS contracts. Following [22] and [13], we define a QoS contract on QoS properties (*QoSProperty*). For each property, a set of service level objective obligations (*SLOObligation*) is specified. An SLO obligation establishes (i) the possible context conditions (*contextCondition*) of system execution; (ii) the SLO to be fulfilled (*SLOPredicate*) under these conditions; and (iii) a guaranteeing reconfiguration rule set (*QoSRuleSet*) to be applied in case of SLO violation. The *QoSGuarantor* refers to the system element that should provide the contracted functionality under the specified SLO obligations. The identification and notification of context changes and of SLOs violations is a responsibility of the *QoSMonitor*.

Example 2 (QoS Contract on Confidentiality). Table 3 illustrates the contract on the QoS property of confidentiality for our video-conference system example. The corresponding e-graph representation is given in Fig. 8.

4.4 Component-Based Architecture Reconfiguration Modeling

Having formalized the structural parts of a system in terms of e-graphs, we define the runtime software architecture reconfiguration as an e-graph transformation system. The definition of this reconfiguration system is based on a definition of a reconfiguration rule.

Definition 6 (Reconfiguration Rule). *A reconfiguration rule, p, is a tuple $(L, K, R, l, r, lt, kt, rt)$, where L (left hand side), K (left-right gluing), and R (right hand side) are e-graphs, and l, r, lt, kt, rt are graph morphisms, abbreviated, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and $lt : L \rightarrow CBS$, $kt : K \rightarrow CBS$ and $rt : R \rightarrow CBS$. p is said to reconfigure L into R.*

Table 3. QoS contract example on confidentiality for the video-conference system.

System Obligations		
Context Condition	Service Level Objective	Guaranteeing Rule Set
1: $conn\_from\_intranet$	$clearChannel$	R.clearChannel
2: $conn\_from\_extranet$	$confidentChannel$	R.confidentChannel
3: $no\_network\_conn$	$localCache$	R.localCache
Responsibilities		
- System Guarantor:	$System.netComp$[a]	
- Context Monitor:	$System.netComp\_AccessPointProbe$[b]	

[a] The system component providing the network connection under the required QoS conditions.

[b] The designated component to check changes on the system network connection's access points and corresponding confidentiality violations.

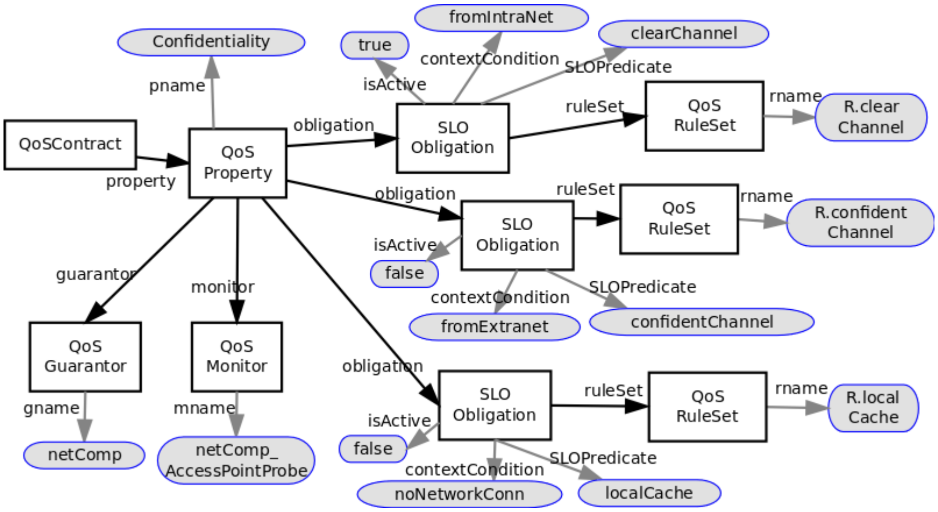

Fig. 8. QoS Contract, in e-graph notation, for the video-conference example. This contract specifies the *netComp* component (cf. Fig. 6) as the *QoSGuarantor*, and an *AccessPointProbe* on this component as the *QoSMonitor* for the confidentiality QoS property. This monitor is used by the system to continually check the changes in the context conditions and violations of the actual SLO. In our example, the initial context condition is a connection *fromIntranet*, and the corresponding SLO is to maintain a *clearChannel*. A context change in connection *fromIntranet* to *fromExtranet* triggers the application of the respective reconfiguration rule set, *R.confidentChannel*. Then, the new context condition would be activated (connection *fromExtranet*). (cf. Tab. 3).

Conceptually, a reconfiguration rule specifies a strategy to address conditions on QoS properties. Thus, for each guaranteeing rule set specified in the QoS contract, associated to a context condition on a given QoS property, the user can encode architectural patterns that address that condition in the left and right hand sides of the rules. Different left hand sides for a similar right hand side in a rule set for a given condition are possible, since the system structures depend on the different context conditions. All left-hand sides of rules in a rule-set are named after that rule-set name. In the scenario of our example, for instance, it is possible to change to a connection from the extranet by moving either from the intranet or from a state with no network connection. Each of these two conditions requires its own system structure, namely, a clear-channel or a local-cache structure, respectively.

Example 3 (Reconfiguration rule). The QoS contract on confidentiality for our video-conference example specifies a guaranteeing set of reconfiguration rules, *R.confidentChannel*, to address the context change when the user moves to the extranet and the contract is violated. Figure 9 illustrates the rule (in that set) that applies when the user is moving from the intranet.

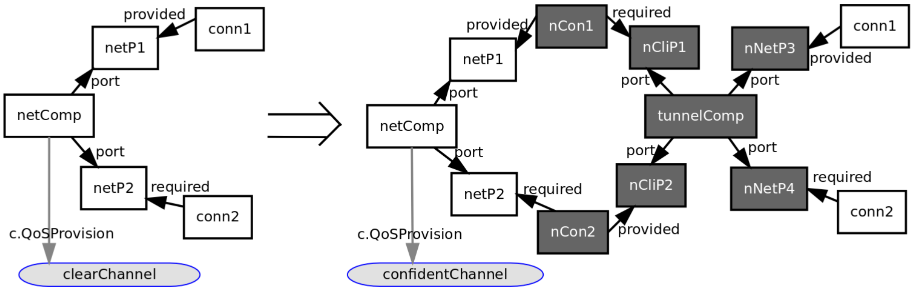

Fig. 9. The $R.confidentChannel$ reconfiguration rule, in e-graph notation, that applies when moving from an intranet network connection to an extranet connection. The left-hand side (LHS) of the rule is used by a pattern-matching algorithm to find a component *netComp* in the system, such that it supports a *clearChannel* as SLO obligation (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 connectors, except those for $conn1$ and $conn2$; (ii) the dark elements must be configured and deployed to provide a tunneled (i.e., confident) channel for the data; (iii) the new ports $nCon1, nCon2$ must be connected to the previously existing ports $netP1, netP2$, and $conn1, conn2$ reconnected to the new ports $nNetP3, nNetP4$, respectively; and (iv) the *c.QoSProvision* attribute of *netComp* must be updated as provisioning a *confidentChannel*. For clarity, the left-right gluing K and graph morphisms l, r, lt, kt, rt are omitted in this figure; K, l, r would correlate each of the corresponding non-dark elements in the RHS with their LHS's counterparts.

Definition 7 (Reconfiguration System). *A component-based reconfiguration system is a tuple $(DSig, CBS, S, C, P)$, where $DSig$ is a suitable data type signature for component-based systems, CBS the component-based structure definition (Def. 3), and S the structure of the system to reconfigure in its initial state, C a QoS contract, and P a set of reconfiguration rules (with S, C and P according to Def. 4, 5 and 6, respectively) in which:*

1. *(When to reconfigure) A system reconfiguration is triggered whenever the QoSMonitor specified in the contract C, $C.monitor$, notifies of an event that violates the actual SLO ($C.property.obligation.SLOPredicate$). This event signals that a new context condition, related to another $C.property.obligation.$ $contextCondition$, is currently in force. Associated to this new context condition, the contract specifies the corresponding SLO and guaranteeing reconfiguration rule set $P = C.property.obligation.ruleSet$.*
2. *(How, Where and What to reconfigure) The identified rule set P is applied to the system reflection structure R_S of S. That is, for each reconfiguration rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in P, and morphism $m : L \rightarrow G$ (called a match of the left-hand side of p, L, in G), we identify a direct reconfiguration $G \overset{p,m}{\Rightarrow} H$ as an e-graph transformation of G into H, as specified by the reconfiguration rule p, of L into R, according to Def. 6.*
3. *A one-step system reconfiguration is a sequence of direct transformations $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$, written $G_0 \overset{*}{\Rightarrow} G_n$, until no more rules in P can be applied.*
4. *The system reconfiguration finishes with a new e-graph reflection system structure, R'_S. The list of actions to reconfigure R_S into R'_S can then be applied to the actual runtime system through f_S^{-1}, according to Def. 4.*

Example 4 (System reconfiguration). Figure 10 illustrates the reconfigured runtime system structure having applied the reconfiguration rule of Example 3 (to be used when the network connection changes from the intranet to the extranet).

5 QoS Contracts-Based Reconfiguration Properties

In this section we analyze the properties of our proposed reconfiguration system as a result of the formalization presented in the previous section.

5.1 Component-Based Structural Compliance

Definition 8 (Full CB-Structural Compliance). *A runtime system reflection structure, RS, is full CB-structural compliant if it is a component-based structure (i.e., if there exists a graph morphism $t : RS \rightarrow CBS$), and the following conditions hold [3]:*

[3] Multiplicity constraints, as defined as usual in CBSE, are omitted for space.

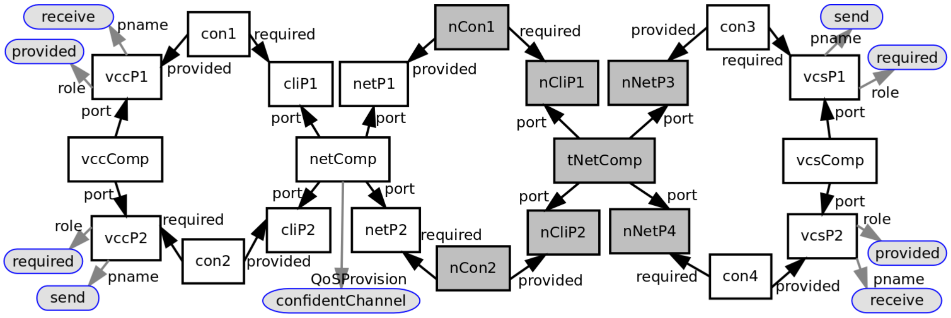

Fig. 10. Reconfigured system architecture in e-graph notation. This new system structure fulfills the SLO (*confidentChannel*) for the new context condition (network connection *fromExtranet*), as specified in the contract illustrated in Fig. 8. The added components are highlighted (shaded). Further details omitted for clarity.

1. $\forall c(c \in RS.Connector \implies \exists p, q(p, q \in RS.Port \implies c.provided = p \wedge c.required = q \wedge c.provided \neq c.required))$*: the ports referenced by the provided and required attributes must be different in every connector.*
2. $\forall p((p \in RS.Port \wedge p.role = Required) \implies \exists c(c \in RS.Connector(c.required = p)))$*: all required ports must be connected.*
3. $\forall c1, c2(c1, c2 \in RS.Connector \implies ((c1.name = c2.name \wedge c1.provided = c2.provided \wedge c1.required = c2.required) \implies c1 = c2)$*: every connector must connect different elements.*

The verifiability of full CB-structural compliance obviously results from the structural definitions 3 and 4 of our reconfiguration system proposal. Even though it would be desirable to statically check that reconfiguration rules produce only full CB-structural compliant systems, this would require more constraints on the reconfiguration rules.

Example 5 (Full CB-structural compliance). The system reflection structures of Fig. 6 and Fig. 10 are full CB-structural compliant, as it is straightforward to verify that the corresponding conditions hold on them.

5.2 Termination and Confluence of the System Reconfiguration

In [10] the *Local Church-Rosser, Parallelism* and *Concurrency* theorems, which hold for graph rewriting, are proved as valid also for typed attributed graph transformation systems. In this section, we show that the one-step system reconfiguration (i.e., $G_0 \overset{*}{\Rightarrow} G_n$ in Def. 7) of our component-based reconfiguration system is reducible to a typed attributed graph transformation system. Therefore, those theorems are also valid for our reconfiguration system.

Theorem 1 (Reducibility of One-Step System Reconfiguration). *Let CBR be a component-based reconfiguration system. A one-step component-based*

(CB) system reconfiguration, $CBSR$, in CBR, is reducible to a typed attributed graph transformation system, $TAGTS$.

Proof. According to Def. 7, a component-based reconfiguration system is a tuple $(DSig, CBS, S, C, P)$. Of these elements, for one-step system reconfiguration (i.e., $G_0 \overset{*}{\Rightarrow} G_n$), the data signature, $DSig$, the component-based structure definition, CBS, and the QoS contract, C, are unchanged. Therefore, in a one-step system reconfiguration these elements can be omitted, depending only on the system reflection structure, S, and the set of reconfiguration rules, P. Given that

1. a CB system reflection structure is a tuple (G, f_S, t), where G is the e-graph that represents a system S through the one-to-one function $f_S : S \to G$, and t is an e-graph morphism $t : G \to CBS$. In the one-step system reconfiguration, f_S also is unchanged and CBS is a type e-graph for G, G attributed with the data signature $DSig$;
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, $u : AG \to ATG$, where ATG is a type graph:
3. a CB reconfiguration rule, p, is a tuple $(L, K, R, l, r, lt, kt, rt)$, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and $lt : L \to CBS$, $kt : K \to CBS$ and $rt : R \to CBS$;
4. the typed attributed graph transformation rules are graph rewriting productions $q = (X \xleftarrow{x} Y \xrightarrow{y} Z)$, X, Y, Z graphs; and
5. both, the system reflection structure and the typed attributed graph are based on the same e-graph definition,

a one-step system reconfiguration, $CBSR$, can be reduced to a typed attributed graph transformation system, $TAGTS$, by making $TAGDSIG = DSig$, $AG = G$ and $ATG = CBS$. The $TAGTS$ set of transformation rules can be defined as the set of CB reconfiguration rules without the lt, kt, rt morphisms, given that, once defined the CB reconfiguration rules, these morphisms are no longer required. □

As a result, the *Local Church-Rosser, Parallelism* and *Concurrency* theorems can then be used with critical pair checking in a particular set of reconfiguration rules, and determine if the one-step system reconfigurations in our reconfiguration system is terminating and confluent. This verification ensures the reliability of the reconfiguration process and frees a system architect of being aware of (i) rule dependencies that may cause deadlocks in the reconfiguration; and of (ii) the rule application order and the specific procedure to perform the reconfiguration itself.

5.3 Stabilization and Exception in the Reconfiguration Process

Given that the reconfiguration rules in our proposal are specified by the user, our reconfiguration system must also consider exceptional cases. These cases correspond to two contract-unfulfilled states, namely the unstable and the exception.

The unstable state is reached when a plausible reconfiguration rule has been found and applied in the system reflection structure, but its effect has not been enough to restore the contract validity. Operationally, in this state the user must be notified about the inefficacy of the rules specified in the contract, after applying the rules a given number of times. On the other side, the state of exception is reached when the reconfiguration system has not been able to find a matching rule to apply in the running system reflection structure. In this case, the user must be notified about the context condition under which the system reflection structure has no corresponding reconfiguration rule, as specified in the contract.

6 Related Work

Software contracts can be seen as a form of property preservation, being this is a recurrent problem in computer science. This problem has been addressed by different communities with different approaches, being a fundamental characteristic of mature engineering disciplines [1]. Our work has been inspired by the general framework approach of some of these proposals, addressing QoS contracts violation through system reconfiguration in component-based systems.

At least in abstract, many of these proposals follow the *rescue* clause idea of the Eiffel's design by contract theory [16]. For example, even though not on the CBSE nor addressing QoS contracts, but on the formal-based self-healing properties preservation side, in [11] Ehrig et al. used algebraic graph transformations for the static analysis and verification of specific properties. Their proposal use a fixed set of particular transformation rules to be applied in response to system failures, thus the self-healing properties are proven with them. Our proposal differs to theirs in that we want to provide a general framework, in the context of component-based software engineering, to be parametrized with reconfiguration rules given by the user; this means that they can prove specific properties, meanwhile we provide tools to the user for checking general properties. Another approach, yet non-formal, aiming at preserving system structural properties in software reconfiguration is the proposed by Hnětynka and Plášil in [12]. Their approach limit the system reconfigurations to those matching three specific *reconfiguration patterns* in order to avoid the dynamic reconfiguration to introduce system architecture inconsistencies.

On the treatment of contracts, in [6] Chang and Collet focuses on the problem of combining low-level properties of individual components to obtain system-level properties as a support for contract negotiation. Their approach identifies then compositional patterns for non-functional properties. On another side, Cansado et al. propose in [5] a formal framework for component-based structural reconfiguration and gives a formal definition of behavioural contract. Their approach is based on a labeled transition system as a formalism to unify behavioural adaptation and determine if a reconfiguration can be performed. Our proposal, even though also address system-level contracts as the two above mentioned, differs to those in that we are interested in the related problems of system architecture and the dependencies on the execution context, meanwhile those deal with more low-level component problems of property composability and interface adaptability, respectively.

7 Conclusions

The main challenge we face in this paper is how to make component-based systems QoS-contracts responsible under varying conditions of context system execution.

In order to face this challenge, we propose a formal approach based on *e-graphs* for system reflection modeling, QoS contract modeling and system architecture reconfiguration. With these definitions, we prove that the one-step system reconfiguration of our component-based reconfiguration system is reducible to a typed attributed graph transformation system. In [10] the *Local Church-Rosser, Parallelism* and *Concurrency* theorems are proved for typed attributed graph transformation systems. Therefore, the adoption of e-graphs to build our component-based transformation system represents three important benefits, as it allows us to: (i) take advantage of the properties of termination and confluence that these theorems allow to check, as a sound strategy for the development of *rule-based*, dynamic, autonomous and self-reconfiguring systems; (ii) provide a rich expressive notation by combining and exploiting graph visual presentations with graph-based pattern-matching; and (iii) benefit from the existing catalogs of design patterns that target different architecture and QoS concerns, as far as the users encode them as reconfiguration rules. In this latter case, our approach enables users to effectively reuse these software design artifacts to enforce particular QoS attribute conditions. For this, however, a more legible and usable concrete syntax should be developed, with automated tools to assist the user in the writing of reconfiguration rules in a more familiar notation such as the used in component-based specifications.

Our formal framework can be used thus to develop and implement rule-based systems in automated and safe ways, being them QoS contracts responsible. With these systems, a user is enabled to define her own rules while freeing her of being aware of the rule application order and of the details of the specific procedure to apply them. For this, and as a result of the formal definition of the QoS contract, component-based systems are enabled as self-monitoring. To this respect, QoS addressing proposals usually detect and manage contract violation either at a coarse-grained, system resources level or at the fined-grained component interfaces level. Our approach is an intermediate proposal, as it takes into account the software components but at the architecture level. Thus, the conditions on QoS properties that we can address can be measured from system context components, and the corrective actions in response to their violation are also at the component-architecture reconfiguration. Nonetheless, from a general point of view, it is possible to formalize in our proposal the global behaviour of the reconfiguration system, defining more precisely the meaning of the contract-unfulfilled states of un-stability and exception, for instance using ideas from process algebras. As future work our plan is (i) to continue the development of our formal framework to form a comprehensive theory for the treatment of QoS contracts in component-based software systems; and (ii) implement it and apply it in representative cases of study to have a better understanding of the different kind of properties that the engineering of self-adaptive software systems must address.

Acknowledgments. This work was funded in part by the Icesi University (Cali, Colombia), the Ministry of Higher Education and Research of Nord-Pas de Calais Regional Council and FEDER under Contrat de Projets Etat Region (CPER) 2007-2013, and during the tenure of an ERCIM "*Alain Bensoussan*" Fellowship by the third author.

References

1. Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Technical Concepts of Component-Based Software Engineering. Volume 2. Technical Report CMU/SEI-2000-TR-008, CMU/SEI (2000)
2. Barbacci, M., Klein, M.H., Longstaff, T.A., Weinstock, C.B.: Quality attributes. Technical Report CMU/SEI-95-TR-021, CMU/SEI (1995)
3. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer 32(7), 38–45 (1999)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. & Exper. 36(11-12), 1257–1284 (2006)
5. Cansado, A., Canal, C., Salaün, G., Cubo, J.: A formal framework for structural reconfiguration of components under behavioural adaptation. Procs. of the 6th Intl. Workshop FACS 2009. ENTCS 263(1), 95 – 110 (2010)
6. Chang, H., Collet, P.: Compositional patterns of non-functional properties for contract negotiation. JSW 2(2), 52–63 (2007)
7. Chang, H., Collet, P.: Patterns for integrating and exploiting some non-functional properties in hierarchical software components. In: Procs. of the 14th IEEE Intl. Conference and Workshops on the ECBS'07. pp. 83–92. IEEE CS (2007)
8. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap pp. 1–26 (2009)
9. Conan, D., Rouvoy, R., Seinturier, L.: Scalable processing of context information with COSMOS. Lecture Notes in Computer Science 4531, 210–224 (2007)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag New York, Inc. (2009)
11. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: FASE'10. LNCS, vol. 6013, pp. 139–153. Springer (2010)
12. Hnětynka, P., Plášil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063. pp. 352–359. Springer-Verlag (2006)

13. Keller, A., Ludwig, H.: The wsla framework: Specifying and monitoring service level agreements for web services. J. Netw. Syst. Manage. 11(1), 57–81 (2003)
14. Krakowiak, S.: Middleware architecture with patterns and frameworks (2009), `http://sardes.inrialpes.fr/~krakowia/MW-Book/`
15. Ludwig, H., Keller, A., Dan, A., King, R.P., Franck, R.: Web Service Level Agreement (WSLA) Language Specification (2003), IBM Available Specification
16. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (1992)
17. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
18. Paspallis, N., Rouvoy, R., Barone, P., Papadopoulos, G.A., Eliassen, F., Mamelli, A.: A pluggable and reconfigurable architecture for a context-aware enabling middleware system. Lecture Notes in Computer Science 5331, 553–570 (2008)
19. Ramachandran, J.: Designing Security Architecture Solutions. John Wiley & Sons, Inc., New York, NY, USA (2002)
20. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA'09. pp. 171–180. IEEE (2009)
21. The OSGi Alliance: OSGi Service Platform Core Specification Release 4. Tech. rep. (June 2009), `http://www.osgi.org/Download/Release4V42`, oSGi Available Specification
22. Tran, V.X., Tsuji, H.: A survey and analysis on semantics in qos for web services. Intl. Conf. on Advanced Information Networking and Apps. pp. 379–385 (2009)
23. Zeng, W., Zhuang, X., Lan, J.: Network friendly media security: Rationales, solutions, and open issues. In: Procs. of the 2004 Intl. Conf. on Image Processing (ICIP). pp. 565–568. IEEE (2004)