Dépt. de Formation Doctorale en Informatique
UFR IEEA

Doctorado en Ingeniería
Depto. de Sistemas y Computación

QoS-CARE: A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration

PhD Thesis

presented and defended in public the 28th of May 2012
to obtain the Title of

Docteur de l'Université de Lille 1 - Sciences et Technologies
and
Doctor en Ingeniería de la Universidad de los Andes
(Specialty in Computer Science)

by

Gabriel Tamura

Composition of the Jury

President :	Juan Francisco Díaz, *Universidad del Valle*
Evaluators :	Ernesto Exposito, *Institut National des Sciences Appliquées (INSA)*
	Lionel Seinturier, *Université de Lille I*
	Silvia Takahashi, *University of Los Andes*
Co-Advisor :	Rubby Casallas, *University of Los Andes*
Co-Advisor :	Laurence Duchien, *Université de Lille I*

Lab. d'Informatique Fondamentale de Lille – UMR USTL/CNRS 8022 – INRIA Lille Nord Europe

Mis en page avec la classe thloria.

Contents

List of Figures

List of Tables

To Norha and
Jose Gabriel
to whom I owe moments,
more than just the time,
I used to finish this work.

Abstract

Ubiquitous software services are increasingly pervading the daily activities of common people. In turn, this situation is increasing the users' demands for highly dynamic capabilities of these services to satisfy context-dependent requirements. In the last years, the engineering of self-adaptive software has demonstrated its support for these capabilities and has achieved significant advances. However, self-adaptation theories, models and mechanisms are further required to be trustworthy, extensible and re-usable in order to be incorporated as common assets of the software engineering design process, when appropriate. Most of the existing approaches focus only on one kind of these assets, usually an ad-hoc mechanism, and moreover, the community lacks standardized properties to evaluate these mechanisms in comparable ways.

To address these shortcomings, in this dissertation we develop and implement a formal model to preserve Quality of Service (QoS) contracts in component-based software applications through dynamic reconfiguration. Our contribution is twofold. First, we provide a comprehensive solution strategy that ranges from a formal foundation that concedes trustworthiness to our proposal, to an experimental evaluation that grants practical feasibility to our work. Second, we identify and define properties inherent to self-adaptive software systems that allow comparable assessment on them.

In our first contribution we identify four elements: the *formal model*, the *architecture* of the proposal, its *implementation*, and its *theoretical validation* and *experimental evaluation*. Our *formal model* is based on the feedback-loop model to achieve reconfiguration autonomy. To obtain reliability, we build on a typed and attributed graph transformation theory to define component-based application structures, QoS contracts and reconfiguration rules. These definitions are typing structures that guarantee conformance of the corresponding instances. We specify the dynamic reconfiguration operation with these operations. Our approach benefits from graph transformation theorems to guarantee that the reconfiguration process is terminating, atomic, and verifiable on the component-based well-formation rules. To cope with context unpredictability, we conceive a finite state machine that manages states of both, contract fulfillment, and also unfulfillment. We devise an *architecture* to realize our formal model, which creates and maintains a graph representation of the managed application and the QoS contracts at runtime. Furthermore, this architecture bridges the graph representation structure with the actual running application structure through a pair of functions that maintain the coherence between these two structures. We encode this architecture as an SCA layer for dynamic reconfiguration in an *implementation*, which we deploy and execute in FRASCATI, a multi-scale SCA middleware. We undertake a *theoretical validation* and *experimental evaluation* of our approach, first by verifying the corresponding conditions of five *adaptation properties*, and second, by applying a benchmark on two plausible case studies. From the obtained experimental results we conclude on the practical feasibility of our proposal.

In the second contribution, we aim at filling-in the existing gap in the assessment of the MAPE-K (Monitor, Analyzer, Planner, Executor and shared Knowledge) loop model, which was proposed with no standard properties to compare among its different realizations. Based on an extensive survey of research papers, we characterized common properties inherent to self-adaptive software –*adaptation properties*, and proposed for them suitable mappings to metrics. Among the most important properties are the atomic adaptation, the settling-time, the termination, the structural consistency, the robustness, the resource overshoot and the stability. For the validation of our proposal, we chose the first five of these properties.

With our first contribution, our aim is to advance the software engineering for self-adaptive software systems by providing a comprehensive solution strategy built on a formal foundation. With our second contribution we provide the basis to standardize self-adaptive software properties, useful for their assessment.

Keywords: CBSE, SCA, QoS contracts, dynamic reconfiguration, self-adaptation properties.

Résumé

L'informatique ubiquitaire envahit de plus en plus nos activités quotidiennes. Cela a pour effet que nous sommes de plus en plus demandeurs de services réagissant dynamiquement au contexte dans lequel nous évoluons. Ces services doivent alors s'adapter à l'environnement dans lequel ils sont utilisés, si possible de façon automatique. Ces dernières années, d'importantes avancées dans le développement de systèmes auto-adaptables ont été faites et ont pu être intégrées dans les logiciels. Néanmoins, il est nécessaire d'étudier des modèles formels et les outils associés pour définir au mieux l'auto-adaptation en vue de garantir la fiabilité, l'extensibilité et la réutilisabilité de ces systèmes, ceci dans un objectif de proposer des artefacts communs et bien définis pour les processus de conception de des applications auto-adaptables. La plupart des approches existantes se concentrent sur l'un des artefacts de ces systèmes auto-adaptables, habituellement sous forme d'un mécanisme ad-hoc. De plus, la communauté manque de standards de comparaison permettant l'évaluation de ces mécanismes d'adaptation et leur fiabilité.

Dans cette thèse, pour faire face à ces problématiques, nous proposons un modèle formel permettant de préserver la qualité de service (QoS pour *Quality of Service* en anglais) à l'aide de contrats dans des applications reconfigurables à base de composants à l'exécution. Notre contribution comprend deux parties. Tout d'abord, nous proposons une solution complète allant de la définition d'un modèle formel de reconfiguration d'un système auto-adaptable fiable jusqu'à l'expérimentation, en démontrant l'applicabilité de l'approche dans une plate-forme à composants reconfigurables. Finalement, nous identifions et définissons des propriétés propres aux logiciels auto-adaptables permettant de les évaluer et de les comparer.

Dans notre première contribution, nous identifions quatre éléments principaux : un modèle formel, une architecture de l'approche, une implémentation, et une évaluation sur deux expériences. Notre modèle formel définit un modèle de boucle de contrôle autonome permettant des reconfigurations autonomes. Le modèle propose des propriétés de fiabilité et se base sur la théorie de transformation des graphes typés attribués. Nous utilisons ce type de graphes et de transformations pour définir la structure des applications à base de composants, les contrats de QoS, et les règles de reconfiguration. Ces éléments sont définis par des structures typées qui garantissent que les instances correspondantes sont conformes au modèle. De plus, nous spécifions un ensemble d'opérations pour la reconfiguration dynamique en utilisant ces définitions. Notre approche bénéficie des théorèmes utilisés dans la transformation de graphes, pour garantir que le processus de reconfiguration soit résiliable, atomique, et vérifiable avec les règles structurelles définies dans les applications à base de composants. Pour faire face à l'imprévisibilité inhérente au changement de contexte, nous définissons une machine à états finis que gère à la fois les contrats qui sont respectés ainsi que ceux qui ne le sont pas. Nous développons une architecture pour mettre en œuvre notre modèle formel par l'intermédiaire d'une représentation, au moment de l'exécution, de l'application et des contrats de QoS associés sous forme de graphes. De plus, cette représentation permet la mise en place d'une liaison causale entre les graphes et l'application dont la cohérence est maintenue et garantie à l'aide de fonctions. Nous mettons en œuvre cette architecture sous la forme de composants SCA (*Service Component Architecture* pour son acronyme en anglais) à l'aide du cadre logiciel FRASCATI, une plate-forme intergicielle multi-échelle définie dans l'équipe ADAM. Finalement, nous présentons une évaluation de notre expérience par un test de performance (benchmark) sur deux cas d'étude. Les résultats obtenus nous permettent de conclure l'applicabilité de l'approche proposée.

Avec notre deuxième contribution, notre objectif est de faire face au manque actuel de méthodes de vérification des instances du modèle de boucle de contrôle de type MAPE-K (*Monitor*, *Analysis*, *Planning*, *Execution* et *Knowledge* par son acronyme en anglais). Ce modèle a été proposé sans qu'aucune propriété standardisée ne soit définie et qui pourrait permettre la comparaison des réalisations de cette boucle de contrôle. Nous avons identifié, à partir d'une étude bibliographique conséquente, un ensemble de propriétés inhérentes aux systèmes auto-adaptables. Nous avons proposé un ensemble de propriétés associées à cette étude. Parmi les propriétés identifiées les plus importantes, nous pouvons citer l'adaptation atomique, le temps de configuration (*settling-time*), la terminaison, la cohérence structurelle, la robustesse, le dépassement de ressources et la stabilité. Pour valider notre approche, nous avons analysé et vérifié les hypothèses pour les cinq premières propriétés.

Avec notre première contribution, nous proposons des avancées dans le domaine de l'ingénierie logicielle pour les applications auto-adaptables. Avec notre deuxième contribution, nous apportons des éléments tangibles pour la standardisation des propriétés pour l'évaluation des systèmes auto-adaptables.

Mots-clés: CBSE, SCA, Contrats de QoS, reconfiguration dynamique, propriétés d'auto-adaptation.

Part I

Motivation and Context

Chapter 1
Introduction

Contents

During the last years, software services have increasingly pervaded all aspects of everyday life. The increasing possibilities of pervasive computing demands highly dynamic capabilities on software to satisfy context-dependent requirements on varying conditions of system execution. Moreover, these dynamic capabilities are required to be trustworthy and performed opportunely to be acceptable to users. Examples of these capabilities are found in businesses following the Service Oriented Computing (SOC) paradigm [Papazoglou *et al.*, 2007], in which component-based services are continually discovered, (re)composed and consumed at runtime. Component services re-composition varies according to changes in context conditions such as those on network access points, user localization, dates, and even interests associated to given locations (e.g. buying souvenirs in the airport on the date flying back home).

To address the problem implied by the aforementioned requirements on dynamic capabilities for component-based services, we identify two key aspects. On one hand, Quality of Service (QoS) contracts are a natural and effective way for capturing this kind of context-dependent requirements [Beugnard *et al.*, 1999, Keller and Ludwig, 2003, Collet *et al.*, 2005, Krakowiak, 2009, Tran and Tsuji, 2009]. On the other hand, over the last decade the engineering of Self-Adaptive Software (SAS) has demonstrated significant advances for supporting dynamic capabilities to satisfy context-dependent goals, such as self-configuration, self-healing, self-protection and self-optimization [Kephart and Chess, 2003, Kramer and Magee, 2007, Salehie and Tahvildari, 2009]. In fact, these two aspects constitute fundamental pillars in the Components promise of software development and evolution based on the assembly and composition of contract-compliant software components. Yet in Component-Based Software Engineering (CBSE) —also called Component-Based Development (CBD) [Szyperski, 1998, Heineman and Councill, 2001], and Service Component Architecture (SCA) [Beisiegel *et al.*, 2007a, Bell, 2008] these two aspects have been considered with different purposes in multiple ways [McKinley *et al.*, 2004, Zschaler, 2004, Chang *et al.*, 2006, Collet *et al.*, 2007]. However, the Component paradigm still requires comprehensive and sound QoS contract-aware self-adaptation theories, models and mechanisms further trustworthy, extensible and re-usable in order to realize its promise.

More specifically, in the CBSE[1] vision, contracts play a fundamental role as they must capture the functional and extra-functional requirements given by users [Bachmann *et al.*, 2000]. For instance, once a QoS contract is negotiated and specified, a system is built to satisfy it by composing components for the corresponding services. However, at any time during system execution, the software that was configured to satisfy a contracted QoS level (e.g., guaranteeing a throughput of 50 transactions per minute on *non-promotion days*) probably would not satisfy other QoS levels to fulfill when the context conditions change (e.g., when *promotion-days* start around Christmas or Thanksgiving, a throughput of 800 transactions per minute can be required, and so on). Naturally, for every expected context situation, a different QoS level to fulfill must be specified as part of the same contract. Moreover, these conditions and QoS levels can be modified by re-negotiations performed at run-time. Therefore, to maintain a QoS contract satisfied (i.e., to preserve it) at run-time, it is necessary to monitor the system's context changes and act in response to them for fulfilling their corresponding QoS levels.

Guaranteeing the *continuous satisfaction* of functional and extra-functional contracts under changing conditions of system execution is one of the main concerns of the engineering of self-adaptive software systems. To address this concern, the Monitoring-Analysis-Planning-Execution and shared Knowledge (MAPE-K or simply MAPE) model has been adopted as a central concept, inspired by principles of control theory [Kephart and Chess, 2003]. Nonetheless, to advance in the realization of its vision, self-adaptation still requires to overcome the following key challenges, among others: (i) the limitations of manually produced adaptation plans to cope with the dynamic evolution of both context situations and software structures to adapt; (ii) the blurred limits between adaptation mechanisms and managed applications that obscure the analysis of their respective properties and advantages; (iii) the limited management of uncertainty to cope with unexpected context changes; and (iv) the lack of standard properties that limit comparable evaluation of adaptation mechanisms and their effectiveness to achieve adaptation goals [Werner Dahm, 2010].

In this dissertation we (i) provide a comprehensive solution strategy for QoS contracts preservation in component-based software applications by dynamically reconfiguring their architecture (i.e., by deploying/undeploying components, and wiring(binding)/unwiring(unbinding) service interfaces); and (ii) identify and propose inherent properties to self-adaptive software –*adaptation properties*, which can be used to standardize comparable assessment models for this kind of systems. Our comprehensive solution strategy comprises four elements, focused in the planner element of the MAPE-K loop model: a *formal model*, its realization as a *software architecture*, the *implementation* of the software architecture as an SCA layer for dynamic reconfiguration to preserve QoS contracts, and a *theoretical and experimental evaluation* of this solution.

The details concerning the problem addressed by this dissertation and our contributions derived from the obtained solution are presented in the remainder of this chapter, as follows. Section 1.1 explains the problem statement, addressed challenges and corresponding research questions. Sections 1.2 and 1.3 present, respectively, the dissertation goals and the assumptions on which we conceive our work. Sections 1.4 and 1.5 respectively present our contributions and

[1]In the following, we use the acronyms CBSE, CBD, and SCA interchangeably.

derived publications, illustrating also the relationship between the contributions and the dissertation goals. Finally, Section 1.6 explains how the remainder of this dissertation is organized.

1.1 Problem Statement and Addressed Challenges

Considering the context analyzed in the previous section, which intersects component-based software engineering, QoS software contracts, and the engineering of self-adaptive software (SAS) systems, we state the main problem addressed by this dissertation as follows:

> *Given an arbitrary component-based software application and a corresponding QoS contract, preserve the continuous satisfaction of the QoS contract in the software application through its dynamic reconfiguration. The preservation of the QoS contract service level obligations must be performed autonomously at runtime, and especially under varying conditions of the software application execution. The preservation operation itself, that is the dynamic reconfiguration, must be able to be parameterized with rule-based strategies for preserving the contracted QoS properties. Most importantly, this reconfiguration operation must be formally guaranteed in inherent properties of self-adaptive software that ensure its reliability.*

Addressed Challenges

As this problem statement is very general, we constrain it with the key challenges identified previously. Moreover, these challenges help us to identify specific research questions that drive the search for the solution. We first state the challenge related to the SAS properties, as the others are related to it, as follows:

C1: Adaptation mechanisms should be evaluated through comparable and clearly defined standard properties. In addition, the improvement, and even the combination of these mechanisms should be based on the assessment of these properties.

C2: Reconfiguration plans to preserve QoS contracts must be generated automatically and dynamically from parameterized reconfiguration rules defined by users. These plans must consider both the current context conditions and the evolved managed application state against the QoS contract specification. Reliability reconfiguration properties must be guaranteed.

C3: There must exist a clear separation of concerns between the reconfiguration mechanism and the managed software application (i.e, the application subject to the QoS contract), and moreover, between their corresponding properties. Additionally, the elements of the feedback loop model, underlying the MAPE model, must be explicit in the adaptation mechanism.

C4: Uncertainty must be managed robustly with respect to the unpredictability of context changes faced by the managed application, as well as the parameterized reconfiguration rules in the reconfiguration mechanism.

C5: The realization of the reconfiguration mechanism for preserving QoS contracts must be feasible as a software architecture and implementation. This implementation must be executable by existing component runtime platforms with reasonable performance.

To identify the first challenge, we analyze the origins of the MAPE loop model, extensively used in the engineering of self-adaptive systems. Despite the MAPE loop was inspired by the feedback loop reference model from control theory, this inspiration considered neither the properties nor the evaluation mechanisms that there exists to assess the corresponding feedback loop instantiations. As a result, self-adaptive software systems use the MAPE loop as a model for its architecture, but nonetheless, even when most of the properties and evaluation mechanisms used for feedback loops are applicable to this kind of software systems by their own nature, they are not used frequently [Salehie and Tahvildari, 2009, Cheng *et al.*, 2009b, Grassi *et al.*, 2009, Kaddoum *et al.*, 2010, Andersson *et al.*, 2009, Villegas *et al.*, 2011b].

Concerning the second challenge, from the components perspective and following CBSE foundations [Szyperski, 1998, Heineman and Councill, 2001], planners for addressing QoS contracts are traditionally designed bottom-up (i.e., contracts are responsibility of single components) and adaptation plans coded by hand [Collet *et al.*, 2005, Léger *et al.*, 2010, Delaval and Rutten, 2010]. However, as software services are composed of several components, the QoS properties of these services depend on the joint work of these components. Hence, planners must devise context-aware reconfiguration plans that reliably manage an arbitrary number of components with their wiring and bindings, considering all differences between the actual vs. the next configuration states, as a whole [Zeng *et al.*, 2004a]. Thus, writing reconfiguration plans by hand is a difficult and error-prone task that should be performed automatically and guaranteeing critical properties.

For the third challenge, as largely analyzed by [Müller *et al.*, 2008] and [Cheng *et al.*, 2009a] among others, the lack of separation of concerns and explicitness of feedback-loop elements in self-adaptive software systems renders their adaptation mechanisms as non-reusable and unanalyzable in their specific properties and advantages. One cause for this problem is the intertwined exploitation of the same realization techniques on both the self-adaptation controller and the managed application, thus blurring their respective properties and limits.

Concerning the fourth challenge, the unpredictable nature of context changes and events that a self-adaptive software must face in its operation is a critical problem still to be solved [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012]. Most of the adaptation mechanisms focus only on a set of foreseen context changes that can affect the operation of their managed systems of interest. However, a robust adaptation mechanism must also manage unforeseen context events, guaranteeing that the properties of the system will remain unaltered even if the managed application system state differs from the expected state. In this setting, an accurate estimation of the safe operational region where the system operates without reaching undesirable states is crucial for the robustness of self-adaptive systems [Meng, 2000, Dowling and Cahill, 2004].

Finally, the fifth challenge is identified in the necessity of evaluating the feasibility and (re)usability of the proposed solution in terms of its actual implementation and performance execution in existing component runtime platforms. Even though many of the proposed solutions (formal or empirical) present implementations, their performance is evaluated by executing them as standalone tools, and for specific managed applications. That is, they are usually not applied in actual component runtime platforms, nor as a generic support independent of the managed application.

Research Questions

It is worth noting two points with respect to the stated challenges. First, as previously mentioned, the self-adaptive software community has no standard properties defined to evaluate this kind of systems, therefore it is necessary to first identify and define them. Second, even with these challenges, the addressed problem is still of considerable size and several interesting research questions arise around it. Thus, in the context of the problem and challenges stated, we identify the following research questions to be addressed in this dissertation:

- What properties are inherent and most significant to mechanisms realizing software self-adaptation?
- Among these properties, which are the most critical for a reconfiguration mechanism to autonomously and reliably preserve QoS contracts?
- How to model and specify QoS contracts to manage and reason precisely about them?
- How to detect when a QoS contract has been violated?
- How to identify the managed application components that need to be reconfigured? How to reconfigure these components and into what?
- What states are needed to control adequately the operation of a managed application subject to satisfy a QoS contract?
- How to manage context uncertainty in QoS contracts, which are context-dependent by their own nature?
- How to achieve a clear separation of concerns between the managed application and the adaptation mechanism, and their corresponding properties?

1.2 Dissertation Goals and Scope

In this section we establish the scope for our solution to the addressed problem by establishing the general and specific goals pursued by this dissertation. Despite choosing specific objectives will not allow us to answer a broader set of related research questions, it will more importantly limit the scope of our work and focus us on finding appropriate solutions for a specific set of them.

Our final aim in this dissertation is to advance in the engineering and assessment of self-adaptive software (SAS) systems through two main goals: (i) characterize inherent properties to SAS systems; and (ii) provide a comprehensive solution for QoS contracts preservation through dynamic reconfiguration, built on a formal foundation. The motivation for having a formal model is to be able to certify desirable properties on the reconfiguration mechanism, as far as possible, or otherwise, at least that the model admits formal analysis on these properties. We refine these two main goals by adopting one specific goal for each of the challenges stated previously, as follows:

G1: Identify and define key properties inherent to software self-adaptation. These properties, such as those for guaranteeing reliability, should serve as candidates for standardization and common basis to evaluate self-adaptation mechanisms.

G2: Develop a formal model for the dynamic and reliable reconfiguration mechanism to preserve QoS contracts in component-based software. This implies for the model to (i) derive reconfiguration plans from high-level reconfiguration rules whenever the contracts are (in risk of) not being fulfilled; and (ii) guarantee critical properties for the reconfiguration reliability.

G3: Maintain a clear separation of concerns between the reconfiguration mechanism and the managed software application, as well as between their corresponding properties. This includes to make the elements of the feedback loop model explicit in the reconfiguration mechanism.

G4: Guarantee robustness in the reconfiguration mechanism with respect to possible and foreseeable situations associated to the management of the unpredictable nature of context. Specifically, to manage context events and conditions, faced by the managed application, that are handled neither by the user-defined event types nor reconfiguration rules in the QoS contracts.

G5: Determine the practical feasibility of the formal based reconfiguration mechanism. We consider it important to analyze and evaluate not only the feasibility of the proposed model in terms of its implementation and performance in plausible scenarios, but also in other practical aspects, such as its (re)usability.

1.3 Assumptions

To fulfill the goals presented in the previous section, we conceive our solution to the problem stated in this dissertation on the following assumptions:

A1: The relationship between specific software design patterns and software quality attributes, given that it has been strongly argued that particular design patterns determine specific levels of software quality attributes [Shaw and Garlan, 1996, Bass *et al.*, 2003, Kruchten *et al.*, 2006, Buschmann *et al.*, 2007, Clements and Shaw, 2009]. Several catalogs of such design patterns have been published in this sense [Ramachandran, 2002, Zeng *et al.*, 2004b, Kircher and Jain, 2004, Krakowiak, 2009, Dougherty *et al.*, 2009].

A2: The theory of assume-guarantee [Inverardi *et al.*, 2009]: as the problem stated requires rule-based solutions, we assume that the user defining the QoS contracts is knowledgeable of design patterns that address the concerned QoS attributes. Even more, that the user will make correct application of these relationships to encode corresponding strategies in the reconfiguration rules. This assumption, combined with assumption $A1$, should be enough to conclude on the plausible efficacy of reconfiguration rules to preserve the continued fulfillment of QoS contracts.

A3: The existence of SCA-compliant [Beisiegel *et al.*, 2007a] component-based runtime platforms with effective capabilities of introspection and dynamic reconfiguration of the executed application. These capabilities must be available at least in a form of reconfiguration primitives that can be dynamically invoked. Currently, we only know of FraSCAti [Seinturier *et al.*, 2009] as one of such SCA runtime platforms with the enumerated characteristics effectively implemented.

1.4 Contribution Overview

We abstract our contribution in two main parts. In the first, we identify and define inherent properties of self-adaptive software (SAS) systems. These properties have been presented and proposed to the SAS community to initiate open discussion for the analysis on their definitions, adoption and standardization. In the second, we provide a comprehensive solution to this dissertation's stated problem that addresses autonomous and reliable reconfiguration of component-based software to preserve QoS contracts under varying conditions of execution. This solution, whose realization we call QoS-CARE (QoS Contract-Aware Reconfiguration systEm), starts with a formal foundation that supports the trustworthiness of our approach, and ends with its implementation and theoretical and experimental evaluation. This implementation and its positive evaluation results determine the practical feasibility, applicability and (re)usability of our reconfiguration system. We have presented the results of this part of our contributions in [Tamura *et al.*, 2011a], [Tamura *et al.*, 2011b], [Tamura *et al.*, 2012], and [Villegas *et al.*, 2012]; while the corresponding to the first, in [Villegas *et al.*, 2011b] and [de Lemos *et al.*, 2012].

In the first part, we distinguish one *General Contribution (GC)* from the characterization of SAS properties (thus labeled *GC.PC*), and one corresponding *specific contribution* (*PC.1*). In the second part, we have three general contributions: two from the formal model (*GC.FM1* and *GC.FM2*), and one from the architecture, implementation and evaluation (*GC.AIE*); and distributed in them, six specific contributions (*FM1.1, FM1.2, FM1.3, FM2.1, AIE.1, AIE.2*). We describe and organize these contributions in the following subsections, and outline them in Fig. 1.1. In this figure, nonetheless, we include also the dissertation challenges and goals to illustrate their respective relationships.

1.4.1 Self-Adaptive Software Properties

In this part of our contribution, we address the lack of standard properties inherent to self-adaptive software, which should be used for comparable assessment, improvement, and composition of different adaptation mechanisms. For this, we start considering the properties defined in control theory [Meng, 2000, Hellerstein *et al.*, 2004, Jacklin *et al.*, 2004], of course re-interpreting them for self-adaptive software. Then, we complement them with others from representative research works in the SAS community. Furthermore, to make these properties concretely measurable, we analyze common adaptation goals and their relationship to quality attributes and corresponding metrics. The general and specific contributions of this part are the following:

GC.PC: *Properties inherent to self-adaptive software systems characterized and proposed for standardization*. Among the most important properties that we characterize are atomic adaptation, short settling-time, termination, consistency, robustness to context unpredictability, small overshoot, and stability. In particular, for the validation of our reconfiguration system we analyze the first five of these properties, defined as follows:

i. Short settling-time: the time taken by the adaptation mechanism for performing the reconfiguration must be acceptable, considering the application domain. Depending on the adaptation goal, this property can be equivalent to the Mean Time to Repair (MTTR), the most critical factor determining availability and reliability.

ii. Termination: the application of the reconfiguration rules in the managed application and the corresponding derivation of the reconfiguration plan from this application (i.e., the reconfiguration process) are guaranteed to terminate.

iii. Atomicity: the reconfiguration is completed as a whole and successfully, or it fails and is rollbacked completely. Additionally, this property enforces robustness, preventing the managed application to reach undesirable and unsafe execution states.

iv. Structural consistency: the preservation of structural integrity constraints, defined by known specifications (e.g., SCA structural conformance rules), on the managed application after each reconfiguration. This property is critical also to avoid undesired states caused by faulty user-defined reconfiguration rules.

v. Robustness to context unpredictability: deviations of the managed application state from the expected ones, or from unexpected context conditions (including e.g., QoS contract specifications), do not alter the properties of the reconfiguration mechanism.

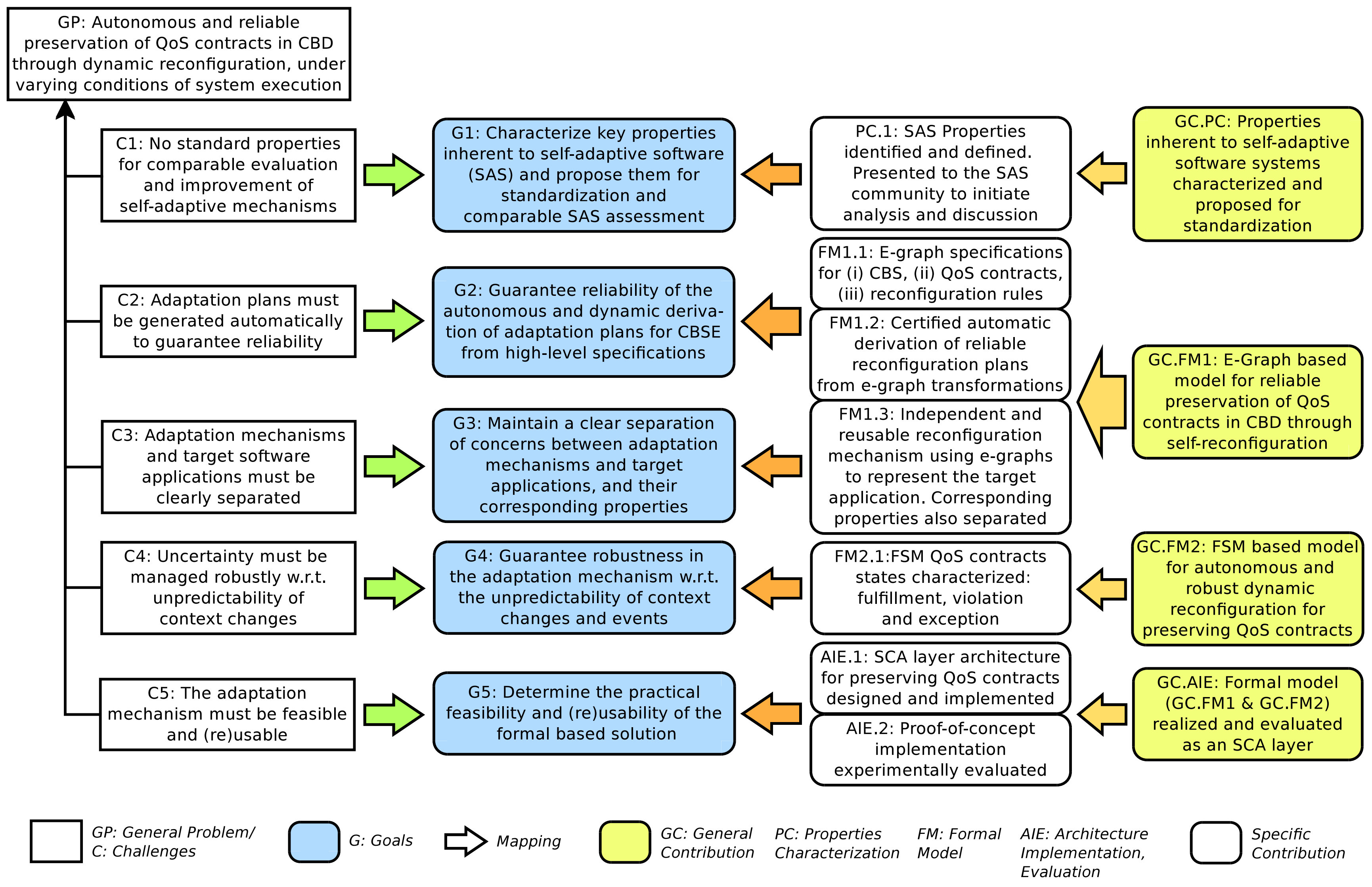

GP: Autonomous and reliable preservation of QoS contracts in CBD through dynamic reconfiguration, under varying conditions of system execution
C1: No standard properties for comparable evaluation and improvement of self-adaptive mechanisms
G1: Characterize key properties inherent to self-adaptive software (SAS) and propose them for standardization and comparable SAS assessment
PC.1: SAS Properties identified and defined. Presented to the SAS community to initiate analysis and discussion
GC.PC: Properties inherent to self-adaptive software systems characterized and proposed for standardization
C2: Adaptation plans must be generated automatically to guarantee reliability
G2: Guarantee reliability of the autonomous and dynamic derivation of adaptation plans for CBSE from high-level specifications
FM1.1: E-graph specifications for (i) CBS, (ii) QoS contracts, (iii) reconfiguration rules
FM1.2: Certified automatic derivation of reliable reconfiguration plans from e-graph transformations
GC.FM1: E-Graph based model for reliable preservation of QoS contracts in CBD through self-reconfiguration
C3: Adaptation mechanisms and target software applications must be clearly separated
G3: Maintain a clear separation of concerns between adaptation mechanisms and target applications, and their corresponding properties
FM1.3: Independent and reusable reconfiguration mechanism using e-graphs to represent the target application. Corresponding properties also separated
C4: Uncertainty must be managed robustly w.r.t. unpredictability of context changes
G4: Guarantee robustness in the adaptation mechanism w.r.t. the unpredictability of context changes and events
FM2.1:FSM QoS contracts states characterized: fulfillment, violation and exception
GC.FM2: FSM based model for autonomous and robust dynamic reconfiguration for preserving QoS contracts
C5: The adaptation mechanism must be feasible and (re)usable
G5: Determine the practical feasibility and (re)usability of the formal based solution
AIE.1: SCA layer architecture for preserving QoS contracts designed and implemented
AIE.2: Proof-of-concept implementation experimentally evaluated
GC.AIE: Formal model (GC.FM1 & GC.FM2) realized and evaluated as an SCA layer
GP: General Problem/ C: Challenges
G: Goals
Mapping
GC: General Contribution
PC: Properties Characterization
FM: Formal Model
AIE: Architecture Implementation, Evaluation
Specific Contribution

Figure 1.1: Mapping among dissertation challenges, goals and contributions

The specific contribution corresponding to this general contribution is the following:

PC.1: SAS inherent properties identified and defined. These properties were presented to the SAS community in papers and discussed in conference events, initiating their analysis towards their standardization.

1.4.2 Formal Model

We conceive our formal model for QoS contracts preservation through dynamic reconfiguration by combining two formal systems: the theory of Finite State Machines (*FSM*) [Hopcroft *et al.*, 2006] and the Typed Attributed Graph (called e-graphs) Transformation System (*TAGTS*) theory [Ehrig *et al.*, 2009]. More precisely, even though this combination is inspired by the MAPE loop [Kephart and Chess, 2003] to achieve reconfiguration autonomy in response to context changes that violate QoS contracts, we use these two formal systems to specifically model the MAPE *Planner* element. The general and specific contributions of our formal model are:

GC.FM1: *E-Graph (TAGTS) based model for reliable preservation of QoS contracts through dynamic reconfiguration*. To obtain reconfiguration reliability in our model, we build on the TAGTS theory to define the structure of component-based applications, QoS contracts and reconfiguration rules. We use these formal definitions as typing structures to guarantee conformance of the corresponding instances, and to specify the dynamic reconfiguration operation with them. Consistently with respect to the current context conditions, the (evolved) managed application state and the QoS contract specification, our model generates reconfiguration plans from the application of parameterized reconfiguration rules defined by users. That is, we abstract the component-based software reconfiguration operation as e-graph transformations; then, we apply the obtained reconfiguration plan to reconfigure the managed application structure. This e-graph based reconfiguration model benefits from existing TAGTS theorems and results, allowing us to guarantee atomicity, reconfiguration termination and well-formation of component-based software applications. The corresponding specific contributions are:

FM1.1: Unified e-graph and machine processable specifications for (i) component-based structures (CBS); (ii) QoS contracts to be satisfied; and (iii) reconfiguration rules (to encode design patterns).

FM1.2: Automatic derivation of reliable reconfiguration plans from rule-based e-graph transformations. This includes functions to obtain the e-graph CBS from the actual running component-based managed application, and to instrument e-graph transformation operations back into the managed application. The implemented functions are executed at runtime.

FM1.3: Independent and reusable reconfiguration mechanism modeled using e-graphs as an abstract and neutral representation for the component-based managed application. The reconfiguration mechanism is clearly separated from the managed software application, as well as their corresponding properties. Furthermore, the relevant elements of the MAPE-loop model are explicit in the reconfiguration mechanism.

GC.FM2: *FSM based model for autonomous and robust management of QoS contract states*. To achieve robustness with respect to the unpredictable nature of context and its implications on the managed application states, we design an FSM that generalizes both the desired and the non-desired states and transitions with respect to QoS contract fulfillment. That is, we derive all reachable states of a managed application execution facing all types of context events and conditions from the user-defined QoS contracts. Thus, this FSM allows us to precisely analyze, reason and deal with these states and transitions, whether specified or not by the user-defined event types and reconfiguration rules. The safe operational region of the managed application, with respect to the reconfiguration mechanism, is confined to the FSM states for which the QoS contract conditions, rules and context events were effectively foreseen by the user. In this way, we guarantee the property of robustness with respect to context unpredictability, as defined previously. The corresponding specific contribution is:

FM2.1: Characterization of generic QoS contracts states (fulfillment, violation and exception) and transitions, automatically managed at runtime by the FSM based model. This includes the management of (i) context events not corresponding to the user-specified QoS contract event types; and (ii) the inefficacy (or non-existence) of user-specified reconfiguration rules to cope with the violation of contracted context conditions.

1.4.3 SCA Architecture, Implementation and Evaluation

To realize our formal model, we devise an SCA architecture that creates and maintains a graph representation of the controlled managed application and corresponding QoS contract at runtime. Furthermore, this architecture bridges the e-graph representation structure with the actual running application structure by implementing a pair of functions that maintain the coherence between these two structures. These functions are implementations of the corresponding specifications that result from the specific contribution FM1.2 (cf. Section 1.4.2). We conceive this architecture as an SCA layer for dynamic reconfiguration to preserve QoS contracts, and implemented, deployed and executed it in FRASCATI[2], a flexible and multi-scale SCA middleware [Seinturier *et al.*, 2009]. We use our proof-of-concept implementation of this architecture to preserve QoS contracts in two plausible application scenarios, and perform an experimental evaluation of its performance by executing it in FRASCATI. From the obtained results we confirm the practical feasibility and (re)usability of our reconfiguration system. The concrete general and specific contributions of this part are:

GC.AIE: *Formal model (GC.FM1 and GC.FM2) realized and evaluated as an SCA layer for dynamic reconfiguration.*

AIE.1: SCA layer architecture for dynamic reconfiguration to preserve QoS contracts designed and implemented maintaining the formal model properties.

AIE.2: Formal model's proof-of-concept implementation experimentally evaluated in a real SCA runtime platform; practical feasibility and (re)usability of the reconfiguration system confirmed.

1.4.4 Relationship between Contributions and Goals

The first part of our contribution addresses our first main goal, that is, to identify, define and propose inherent properties to SAS systems as a common basis to evaluate this kind of software systems. Correspondingly, the second part of our contribution addresses our second stated main goal, that is to advance the software engineering for SAS systems by providing a comprehensive solution to preserve QoS contracts in component-based software, based on a formal foundation.

Besides providing an autonomous and reliable SCA layer for dynamic reconfiguration to preserve QoS contracts, QOS-CARE also supports software-evolution architects in the reliability

[2]We used FRASCATI v. 1.4 fixing some bugs found in the execution of reconfiguration primitives

and efficacy analysis of parameterized reconfiguration rules. The SCA-compliant architecture of QOS-CARE explicitly differentiates each of the MAPE-loop elements, maintaining a clear separation of concerns between the reconfiguration mechanism and the managed application. Our modeled planner uses a top-down strategy to address QoS properties allowing the user to encode known design patterns in reconfiguration rules and synthesizing dynamically reconfiguration plans from the application of rules through pattern-matching. From the SCA point of view, QOS-CARE is a complementary layer for software component platforms that provides the capability to preserve QoS contracts for the software executed with them. Nonetheless, QOS-CARE achieves independence of managed applications, and even more, of component platforms themselves, in grace of the mentioned separation of concerns. Additionally, this independence allows us to characterize QOS-CARE comparatively to other adaptation mechanisms with respect to their possibilities of composition or combination, thus leveraging their combined properties in increasingly wider and more complex settings.

Concerning the *adaptation properties*, in this dissertation we consider the reliability of the reconfiguration process defined as the continuity of (agreed) expected service [Avizienis *et al.*, 2004]. Moreover, we define the reliability of QOS-CARE in terms of the five properties enunciated previously: short settling-time, termination, atomicity, SCA structural conformance and robustness with respect to context unpredictability. Following [Candea *et al.*, 2004] and [Hellerstein *et al.*, 2004], we interpret the reconfiguration settling-time as the mean-time to recover (MTTR) metric. Particularly applied to self-reconfigurable systems, settling-time also determines the acceptability of the reconfiguration time, which naturally must be evaluated empirically. SCA structural conformance is a property that must be verified at runtime on the managed application to ensure that its structure conforms to the structural integrity constraints defined in the SCA specification after each reconfiguration [Beisiegel *et al.*, 2007a, Léger *et al.*, 2010]. For the properties of termination and atomicity, we show formally that they are guaranteed by our formal model. Finally, by considering reliability in the sense of being able to be trusted but also measurable, QOS-CARE can be configured to provide system evolution architects with empirical assessments of the MTTR for particular application domains. This empirical assessment, combined with the other properties, can serve as a basic figure of the extent of the confidence that users can have on our reconfiguration system for different managed applications and scenarios. The experiments performed with QOS-CARE integrated in a real SCA implementation and running a reasonable software application (i) constitute an additional proof of our formal model soundness; (ii) allow us to conclude positively on the evaluation of its reliability; and (iii) determine its practical feasibility and (re)usability.

More importantly, in our opinion, the adaptation properties that we propose for standardization help to leverage the benefits of self-adaptation, not only for its wider adoption in industry, but also for the software engineering discipline itself. In the first aspect, for instance, these properties and associated metrics allow users not only to measure but also to compare and improve the reliability and trustworthiness of different adaptation mechanisms. In the second aspect, the corresponding measurements on these properties and metrics could provide dependability insights to incorporate these mechanisms in the partial automation of the maintenance and evolution phases of software life cycles.

1.5 Publications Derived from this Dissertation

The results produced in the development of this dissertation have been published in international journals, symposiums, book chapters and workshops, as follows.

International Refereed Journals

- Gabriel Tamura, Rubby Casallas, Anthony Cleve, and Laurence Duchien. *QoS-CARE: A Formal System for Reliable QoS Contract Preservation in Component-based Software*. Journal Science of Computer Programming (Special Issue in Formal Aspects of Component Software), 2012. (*In Evaluation*) [Tamura *et al.*, 2011b].

Book Chapters

- Rogerio de Lemos, Holger Giese, Hausi Müller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Bradley Schmerl, Dennis B. Smith, João P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. In: Software Engineering for Self-Adaptive Systems 2, 2012. (*In Press*) – This paper is a revised and extended version of the one published in the Dagstuhl 10431 Seminar Proceedings as [de Lemos *et al.*, 2012].

- Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, João P. Sousa, Basil Becker, Mauro Pezzè, Gabor Karsai, Serge Mankovskii, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems*. In: Software Engineering for Self-Adaptive Systems 2, 2012. (*In Press*) [Tamura *et al.*, 2012].

- Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Ruby Casallas. *DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems*. In: Software Engineering for Self-Adaptive Systems 2, 2012. (*In Press*) [Villegas *et al.*, 2012].

International Symposiums

- Norha Villegas, Hausi Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. *A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems*. Procs. of 6th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2011 [Villegas *et al.*, 2011b].

International Workshops

- Gabriel Tamura, Rubby Casallas, Anthony Cleve, and Laurence Duchien. *QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs*. Procs. of 7th Intl. Workshop on Formal Aspects of Component Software, 2010. This paper was selected also for publication on the LNCS Vol. 6921 – Formal Aspects of Component Software [Tamura *et al.*, 2011a].

Electronic Magazines

- Gabriel Tamura and Anthony Cleve. *A Comparison of Taxonomies for Model Transformation Languages*. In: Paradigma – Revista Electrónica en Construcción de Software, 2010 [Tamura and Cleve, 2010].

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows.

Chapter 2: Context and State-of-the-Art Background. The problem we address in this dissertation intersects the engineering of self-adaptive software (SAS) systems, component-based software (CBSE/CBD/SCA) and QoS software contracts. Hence, in this chapter we present a background on the state of the art of these topics with their related research challenges, key concepts, and definitions of terms, as understood in this dissertation.

Part II: Contribution

Chapter 3: Quality-Driven Self-Adaptation Properties. In this chapter we define the properties that we identify as inherent to self-adaptive software (i.e., adaptation properties) and should be used for comparable assessment of adaptation mechanisms. We distill these properties from a survey of representative self-adaptive software research works. Additionally, to make these properties measurable, we analyze common adaptation goals, and their relationship to quality attributes and corresponding metrics. From these properties, in the following chapters we select a subset that coherently supports the fulfillment of the reliability and robustness goals of this dissertation.

Chapter 4: A Formal Model for QoS Contracts-Preserving Reliable Reconfiguration. In this chapter we present our formal model for autonomous and reliable preservation of QoS contracts through dynamic reconfiguration, targeting component-based software. On one side, we give E-Graph definitions for component-based software structures, QoS contracts and reconfiguration rules to specify our formal reconfiguration system to reliably preserve QoS contracts. On the other side, we present the FSM based model for autonomous and robust specification of transitions between the managed application states, characterized with respect to QoS contracts fulfillment and unfulfillment. We show how do we combine these two structures for our formal model to be autonomous and reliable.

Chapter 5: QoS-CARE: The Realization of Our Formal Model. The SCA specification is commonly implemented as a middleware of stacked layers of pervasive functionalities and services for executing component-based software applications. In this chapter we present the architecture, design decisions, and implementation of QoS-CARE, the realization of our formal model. In the context of the SCA stack, we conceive QoS-CARE as an additional platform-independent SCA layer for dynamic reconfiguration to preserve QoS contracts in component-based software applications, maintaining the properties of the formal model.

Part III: Validation

Chapter 6: Validation and Verification of QoS-CARE Properties. In the development of our reconfiguration system we benefit from properties and definitions of both Typed Attributed Graph (E-Graphs) Transformation systems and FSM theories. In this chapter we analyze these properties and interpret them in the context of a selected subset of the adaptation properties presented in Chapter 3. We also illustrate the significance of these properties for guaranteeing the reliability and robustness of our model for dynamic reconfiguration to preserve QoS contracts.

These properties are the short settling-time, the reconfiguration termination, the atomicity, the SCA structural conformance, and the robustness with respect to context unpredictability. For the settling-time we use a benchmark for measuring the Mean-Time to Reconfigure (MTTR). SCA structural conformance and robustness to context unpredictability are verified at runtime as part of the reconfiguration process, supported by our formal model: the first, by the SCA constraints on the e-graph representation of the managed application, meanwhile the second by checking the types of context events, and the efficacy and existence of reconfiguration rules. Atomicity and reconfiguration termination are guaranteed as a result of theoretical properties of our formal model.

Chapter 7: QoS-CARE Validation Scenarios. In this chapter, in addition to the validation and verification of QoS-CARE's properties presented in Chapter 6, we also evaluate the applicability and performance (i.e., the mean-time to reconfigure, MTTR) of our reconfiguration system. We base this empirical evaluation on a set of experiments performed by using QoS-CARE to preserve QoS contracts in two validation scenarios, executed with the FraSCAti SCA runtime platform. For each of these scenarios, we describe the component-based software application, the respective QoS contract, reconfiguration rules, and the obtained experimental data. For the particular reconfiguration rule-sets we analyze the termination and SCA structural conformance conditions, whereas for the experimental results, the mean-time to reconfigure. We then conclude on the practical feasibility, (re)usability and applicability of our reconfiguration system.

Part IV: Summary

Chapter 8: Conclusions and Future Work. In this chapter we summarize the work presented in this dissertation. We highlight our contributions analyzing our overall approach in its advantages, and discussing also its limitations. Additionally, we distinguish the limitations that, in our opinion, deserve attention as worth of future research work.

1.7 Chapter Summary

In this chapter we have presented the problem statement, addressed challenges, goals, assumptions and a contributions overview of this dissertation. On one side, from the problem statement we identified and derived the dissertation addressed challenges. For each of these challenges we adopted one goal, as established in Section 1.2. On the other side, we presented the relationship of fulfillment between our contributions and the stated goals in Section 1.4.4. We also illustrated our assumptions as independent suppositions and inspirational foundations on which we build our reconfiguration system for preserving QoS contracts in component-based software. All of these elements with their respective relationships, which summarize the addressed problem, motivation and contributions of this dissertation, were outlined in Fig. 1.1.

Figure 1.2 abstracts, in the form of a conceptual map, the dissertation overview and the relationships among its challenges, goals and contributions.

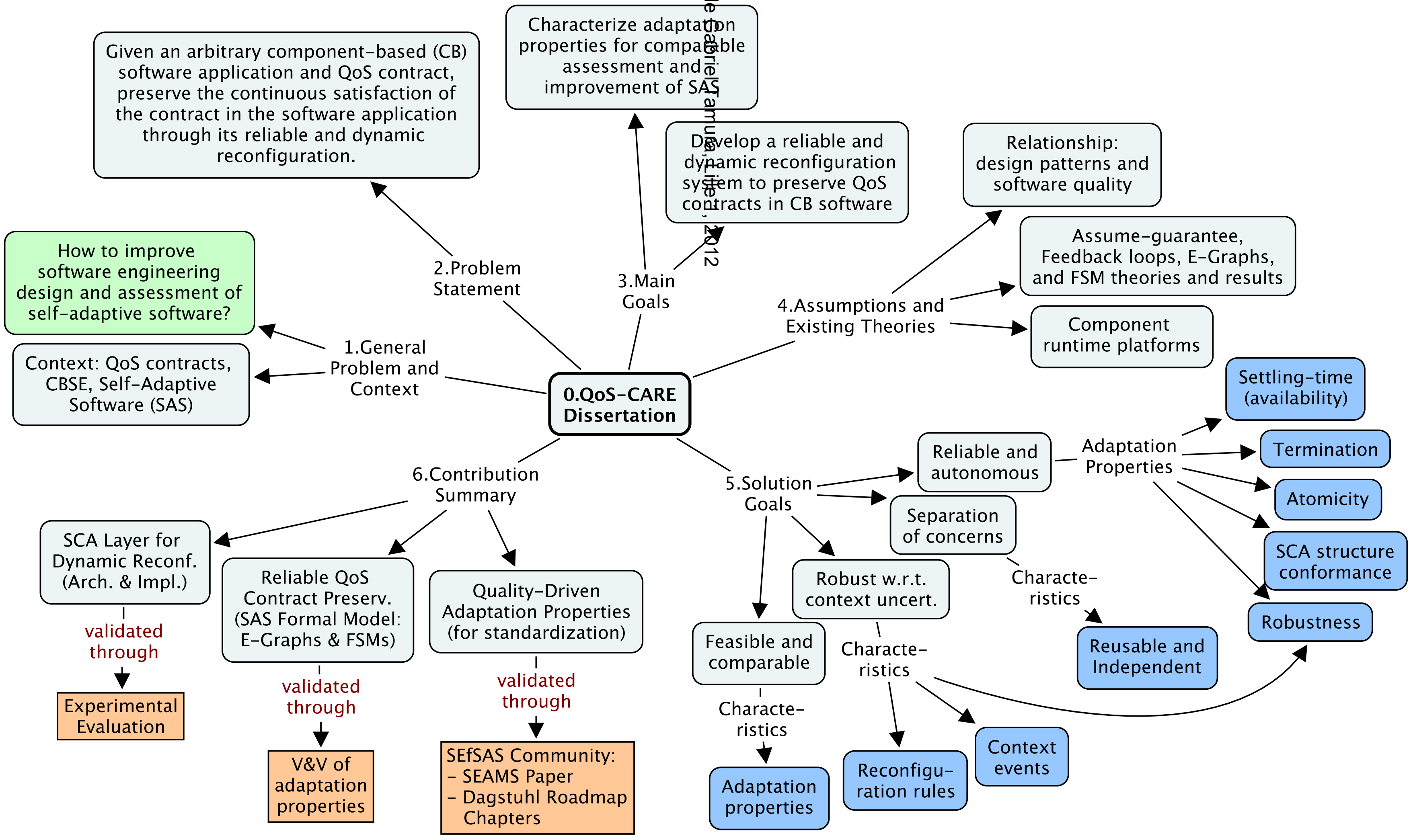

0.QoS-CARE Dissertation
1.General Problem and Context
How to improve software engineering design and assessment of self-adaptive software?
Context: QoS contracts, CBSE, Self-Adaptive Software (SAS)
2.Problem Statement
Given an arbitrary component-based (CB) software application and QoS contract, preserve the continuous satisfaction of the contract in the software application through its reliable and dynamic reconfiguration.
3.Main Goals
Characterize adaptation properties for comparable assessment and improvement of SAS
Develop a reliable and dynamic reconfiguration system to preserve QoS contracts in CB software
4.Assumptions and Existing Theories
Relationship: design patterns and software quality
Assume-guarantee, Feedback loops, E-Graphs, and FSM theories and results
Component runtime platforms
5.Solution Goals
Reliable and autonomous
Separation of concerns
Robust w.r.t. context uncert.
Feasible and comparable
Adaptation Properties
Settling-time (availability)
Termination
Atomicity
SCA structure conformance
Robustness
Characte-ristics
Reusable and Independent
Characte-ristics
Reconfigu-ration rules
Context events
Characte-ristics
Adaptation properties
6.Contribution Summary
SCA Layer for Dynamic Reconf. (Arch. & Impl.)
validated through
Experimental Evaluation
Reliable QoS Contract Preserv. (SAS Formal Model: E-Graphs & FSMs)
validated through
V&V of adaptation properties
Quality-Driven Adaptation Properties (for standardization)
validated through
SEfSAS Community: - SEAMS Paper - Dagstuhl Roadmap Chapters

Figure 1.2: Dissertation overview and relationships among challenges, goals and contributions

Chapter 2

Context and State-of-the-Art Background

Contents

The problem we address in this dissertation intersects the engineering of component-based software systems[3], Quality of Service (QoS) software contracts, and self-adaptive software (SAS) systems. Hence, in this chapter we present the fundamental concepts of each of these three research areas, their current main concerns, key research challenges, approaches and associated technologies to solve them, as related to the scope of this dissertation.

Therefore, in Section 2.1 we present the key concepts on which these research areas are developed, giving our own definitions or interpretations when appropriate. In Section 2.2 we present the foundational concepts of the Component paradigm for software development, and describe representative run-time component platforms with their respective limitations. In Section 2.3 we analyze the different ways of how QoS software contracts have been specified, processed and used in software systems, as well as the strategies utilized for guaranteeing them. In Section 2.4 we present the fundamental concepts of autonomic computing and its foundational ideas and

[3]Also known as Component-Based Development (CBD), Component-Based Software Engineering (CBSE) and Service Component Architecture (SCA), we use these terms interchangeably.

characteristics, namely, self-adaptation, self-healing, self-optimization, and self-protecting. In Section 2.5 we introduce the application scenario that we use along this dissertation to illustrate the definitions of our formal model, and its implementation as an SCA-compliant reconfiguration mechanism.

2.1 Definitions of Terms

In this section we present a list of terms used along this dissertation with their respective definitions, including our own interpretations when appropriate. Most of these definitions are based on the references listed in the Bibliography section, and the Merriam Webster Dictionary (Online version available in `http://www.merriam-webster.com/dictionary`). In particular, the definitions for autonomic computing terms are mainly based on [IBM Corporation, 2006].

CBD/CBSE/SCA: Component-Based software Development/Component-Based Software Engineering/Service Component Architecture. These acronyms are used to refer to the paradigm of software development based on software components [Szyperski, 1998, Heineman and Councill, 2001].

SAS: Self-Adaptive Software system. A software system with the capability of modifying itself at runtime in response to changes in its execution environment. These changes include resource availability variations, modifications on the user needs, intrusions, and faults. The purpose of the adaptation is to preserve a satisfactory operation under different context situations with no (or limited) human intervention [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012]. SAS are composed of two parts: (i) the adaptation mechanism, and (ii) the managed software application.

Adaptation mechanism: (also *reconfiguration mechanism*) the mechanism (theoretical model or software artifact) that performs (in abstract or concrete form) the dynamic adaptation or reconfiguration of the *managed software application* [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012]. In a self-adaptive software system, the *adaptation mechanism* is the counterpart of the managed software application.

Managed software application: (also *managed application*, *managed system*, *target application*) term used for the software application to be adapted or reconfigured in a self-adaptive software system [Hellerstein *et al.*, 2004, Villegas *et al.*, 2012]. In these systems, the *managed application* is the counterpart of the *adaptation mechanism*. In the context of this dissertation, the managed application is also the software application subject to the QoS contract to be satisfied, especially under changing context conditions of its execution.

Dynamic reconfiguration: the operation that modifies, at runtime, the configuration of a software application. If this operation is performed by the same software system, it is called self-reconfiguration. From the software architecture point of view, this means to modify the structural description of the application, given, for instance, in an Architecture Description Language (ADL). In the Component paradigm, the configuration of a software application is expressed in terms of its composites, components, exposed properties, wires and bindings [McKinley *et al.*, 2004, Seinturier *et al.*, 2012].

MAPE-K loop reference model: (also MAPE loop) a reference model proposed by IBM to guide the design of self-managing and self-adaptive software systems [Kephart and Chess, 2003]. Inspired by the feedback-loop model from control theory, this model defines a reference structure for these kinds of adaptive software systems, as composed of five elements: Monitor, Analyzer, Planner, Executor and Knowledge manager. For each element, it assigns specific functionalities and information flow aimed at autonomously controlling a given managed software application.

Quality of Service (QoS) contract: the precise specification of the expected *levels of operation* (QoS levels) that must be guaranteed on a given QoS attribute (e.g., confidentiality, availability, performance) by a managed software application [Collet *et al.*, 2005, Jureta *et al.*, 2009, Tran and Tsuji, 2009]. These QoS levels must be specified for each of the different context conditions to be faced by the managed application in its execution. The continuous satisfaction of a QoS contract (i.e., its preservation) implies to satisfy each of these QoS levels that the user expects, under each of the corresponding varying conditions of system execution.

Reliability (classic definition): the *continuity of expected* service delivery. Any deviation from the service delivery in the way it is agreed is considered a reliability violation [Avizienis *et al.*, 2004]. Reliability has been traditionally measured in relation to the response that a system exhibits to its own *failures*, that is, in the sense of system trustworthiness [Reussner *et al.*, 2003, Candea *et al.*, 2004, Yacoub *et al.*, 2004, Filieri *et al.*, 2010, Huang *et al.*, 2011].

Reliability (our definition): the *continuity of expected QoS-levels fulfillment* in software application services when facing not only natural context changes, but also unexpected anomalous situations (e.g., faulty reconfiguration rules given by users).

Target application: (also *target software application*) synonym of *managed software application*.

Target system: term used in the control engineering domain for the system to be controlled (e.g., usually a physical plant for an industrial process) [Ogata, 1996, Hellerstein *et al.*, 2004]. In the SAS domain, *target system* corresponds to the *managed (software) application* or *target (software) application*.

2.2 Component-Based Software Engineering

In the last ten years, the Component paradigm for building software systems has evolved based on a fundamental vision of the software component as a contract or obligations-responsible software artifact.

In this paradigm, software units are encapsulated in *components*. A component is a gray-box software artifact with well-defined *services* (or *interfaces*), and exposed properties. A component *required* service can be satisfied by a *provided* service offered by another component, as far as these services implement the same *interface*. To exchange information among them, components communicate either by wiring required- to provided-services directly, or by binding them to communication protocols such as SOAP, RMI, JMS or REST. Components can contain other components hierarchically (thus called *composite* structures) and, ideally, can be implemented using different programming languages. Another claimed advantage is that defining a precise set of contractual obligations allows components to be independently developed and deployed, guaranteeing that, if they realize their responsibilities, they can interact in expected ways [Szyperski, 1998, Bachmann *et al.*, 2000, Heineman and Councill, 2001].

In light of this vision, the Component paradigm has been used for engineering software systems in a wide variety of forms. These forms include building systems from contract-compliant components, to abstracting reflection mechanisms at the component-level (i.e., composite, component, interface, binding) to support the adaptation of self-managing software systems at runtime. Nonetheless, although significant research has been conducted on guaranteeing functional and extra-functional contracts by software components systems, the autonomous preservation of QoS contracts under varying conditions of execution is known as a research problem still to be solved. We analyze this in the next sections.

2.2.1 Component Models

To realize the Component paradigm's vision and promises, several component models have been developed and implemented, each with a different syntax, but all of them sharing the same basic concepts. Examples of these models that several years ago transitioned into the industry are OMG's CORBA Component Model (CCM[4]), Oracle's (formerly SUN) Enterprise Java Beans (EJBs[5]), Microsoft's Component Object Model (COM, DCOM and variants[6]). For the graphical representation (i.e., the syntax), most of these models follow the OMG's UML component diagram specification[7]. An important variation of flat component models was introduced by hierarchical ones, such as the presented in FRACTAL [Bruneton *et al.*, 2006]. Hierarchical component models address scalability and abstraction issues by allowing components to be defined as composed of other components, sharing of components (e.g., to model shared resources), and providing reflection capabilities. In fact, reflection is the most important requirement not only for self-monitoring but also for dynamic reconfiguration. However, despite the many important achievements, concerning the autonomous preservation of QoS contracts these models have not completed their evolution to realize the initial vision of the Component paradigm.

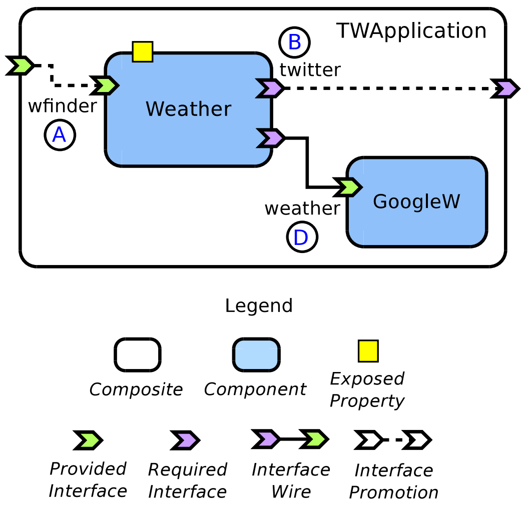
2.2.2 The Service Component Architecture (SCA) Specification

More recently, several IT providers joined efforts and formed the so-called Open Service Oriented Architecture (OSOA) Collaboration with the purpose of defining a language-independent programming model for building and executing Service Oriented Architecture (SOA) distributed applications. In this way, the specifications given by the first component models, which were proposed by some of the OSOA founders, served as a base to develop the Service Component Architecture (SCA) set of specifications. Hence, these specifications adhere to the previously enunciated component-based definitions and principles.

The SCA core specification, which is the component assembly language, defines a model that is also independent of Interface Definition Languages (IDLs), communication protocols, and non-functional properties [Beisiegel *et al.*, 2007a][8]. Thus, SCA is aimed at leveraging the wide range of existing technologies for implementing software components and its services (e.g., Java, Python, Scala, and PHP), and for connecting them (through e.g., Web services, SOAP, REST, RPC, and RMI). In Fig. 2.1 we illustrate the SCA notation in its graphical (left), and XML-based assembly language (right) representations. In this figure, the `TWApplication` composite is composed of two components, `GoogleW` (Google Weather information provider) and `Weather`. From this latter, the application composite promotes (i.e., makes externally visible) the `wfinder` provided service (labeled (A) in the figure), and the `twitter` (B) required service. As illustrated in the assembly definition, this service is bound to the Twitter REST service (C)[9] used for retrieving

[4]http://www.omg.org/spec/CCM/4.0/PDF/06-04-01.pdf
[5]http://www.oracle.com/technetwork/java/ejb-141389.html
[6]http://www.microsoft.com/com/default.mspx
[7]http://www.omg.org/spec/UML/2.2/Superstructure
[8]http://www.oasis-opencsa.org
[9]http://twitter.com

the location of the user specified in the `userId` exposed property, from the Twitter profile. The `GoogleW` component uses this information to retrieve the weather conditions in this location through the `weather` provided service (D).



TWApplication.composite

```
<composite name="TWApplication">
  <service name="wfinder" promote="Weather/wfinder">
    <binding.ws uri="http://192.1.1.2:9091/TWFinder"/>
  </service>
  <component name="Weather">
    <implementation.java
        class="qosCaredApps.tw.WeatherImp" />
    <service name="wfinder"/>
    <reference name="twitter">
      <binding.rest uri="http://twitter.com"/> C
    </reference>
    <reference name="weather" target="GoogleW/weather"/>
    <property name="userId">gabriel_tamura</property>
  </component>
  <component name="GoogleW">
    <implementation.java class="qosCaredApps.tw.GWImp"/>
    <service name="weather">
      <interface.java
          interface="qosCaredApps.tw.WeatherIfc"/>
    </service>
  </component>
</composite>
```

Figure 2.1: The SCA notation in its graphical (left), and XML-based (right) representations. Components provide services (through interfaces) that can be required by others. Connections from required to provided services (interfaces) specify the corresponding service invocations.

The SCA specification has been implemented by several IT vendors and research projects, as described in the OpenSOA site[10]. Examples of these implementations are IBM's Rational Application Developer/WebSphere Application Server for SCA[11], Oracle's SOA/EDA[12], Apache's Tuscany[13], Fabric3[14] and FRASCATI[15], among others.

This specification is currently maintained by OASIS[16], who adopted it for formal standardization under the name of Open Composite Services Architecture (Open CSA). Additionally, SCA has been identified as the foundation for yet other promising computing paradigms, such as the Service Oriented Computing [Papazoglou *et al.*, 2007] and Cloud Computing [Merle *et al.*, 2011].

Support for Non-Functional Requirements

As related to the goal of this dissertation, the SCA specification supports non-functional requirements (e.g., security, transaction, logging) on services through *policy sets*. These are defined by the SCA Policy Framework [Beisiegel *et al.*, 2007b].

The Policy Framework is defined in two levels. The first, called the abstract level, defines *policy intents* as high-level language constructs for expressing non-functional requirements for component services. Services with this kind of requirements are annotated with tags in order to declare the corresponding policies. For instance, a policy intent can specify that a given service

[10]http://www.osoa.org/display/Main/Implementation+Examples+and+Tools
[11]http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca
[12]http://www.oracle.com/us/technologies/soa/index.html
[13]https://cwiki.apache.org/TUSCANY/sca-java-2x-releases.html
[14]http://www.fabric3.org
[15]https://wiki.ow2.org/frascati
[16]http://www.oasis-opencsa.org

requires of authentication prior to its use (e.g., with the `@Authentication` tag). As this is an abstract specification, a low level configuration must be also specified to solve platform-specific details at deployment time.

The second level, called the concrete policy, specifies the low-level configuration. This configuration is used at deployment time by mapping intents to specific policy choices. In the example of authentication, the concrete policy set could specify the options and preferences for using different authentication methods, such as Kerberos, MD5, and Diameter. At deployment time, the SCA implementation is responsible for enforcing the policies, for instance allowing the selection of one of the specified alternatives in a policy set.

Intents and policy sets may be applied to services (interaction policies), as well as to component implementations (implementation policies). However, although this mechanism is a first step to address non-functional properties in components and their services, it has some limitations. In particular, it was not designed to fulfill varying QoS-level objectives autonomously, that is, in response to the dynamic system execution's changing conditions. These requirements are addressed by the engineering of context-aware and self-adaptive software systems, which we discuss in Section 2.4.

2.2.3 The FraSCAti SCA Implementation

FraSCAti is an open source and multi-scale implementation of the SCA specification for developing and executing distributed SCA applications. In the form of a middleware software stack, it adds reflection capabilities to the standard SCA functionalities. Thus, these capabilities of introspection and primitive reconfiguration (i.e., adding/removing components, wires, and bindings) can be exploited as pervasive services by any SCA application executed in FraSCAti [Seinturier *et al.*, 2009, Seinturier *et al.*, 2012].

As illustrated in Fig. 2.2, the FraSCAti architecture is an SCA stack with four layered levels:

1. The Kernel Level (in the bottom part of the figure) is based on the Fractal component model [Bruneton *et al.*, 2006]. This is a lightweight and open component framework that implements the notion of component by exploiting dependency injection, introspection and primitive reconfiguration capabilities. The model is inspired on the ideas of software architecture [Shaw and Garlan, 1996], and distributed, re-configurable and reflective systems [Smith, 1984].

In particular, the reflection capabilities of the model provides meta-level operations over the components structure using control interfaces. These operations support the dynamic reconfiguration of components and their interconnections at run-time.

2. The Personality Level. As illustrated in the figure, the personality level adds to the definition of kernel components one controller part and two interceptor parts (one for the provided and another for the required interfaces). The interceptors can be used to modify or extend the behavior of service invocations, whereas the controller allows the configuration of different "facets" of the personality, for instance in terms of specific component life-cycles or binding management. Component life-cycle management includes starting and stopping its execution according to the defined component instance models (e.g., STATELESS, CONVERSATION, COMPOSITE, and REQUEST).

This level also enables the use of primitive reconfiguration as a common functionality of the component platform through the controller interfaces, thus extending the SCA specification.

3. The Run-time Level. This level is responsible for instantiating and executing the SCA composites and components in the runtime platform. In the figure, we illustrate the main composites that comprise this level, as related to the goals of this dissertation. On one hand, the

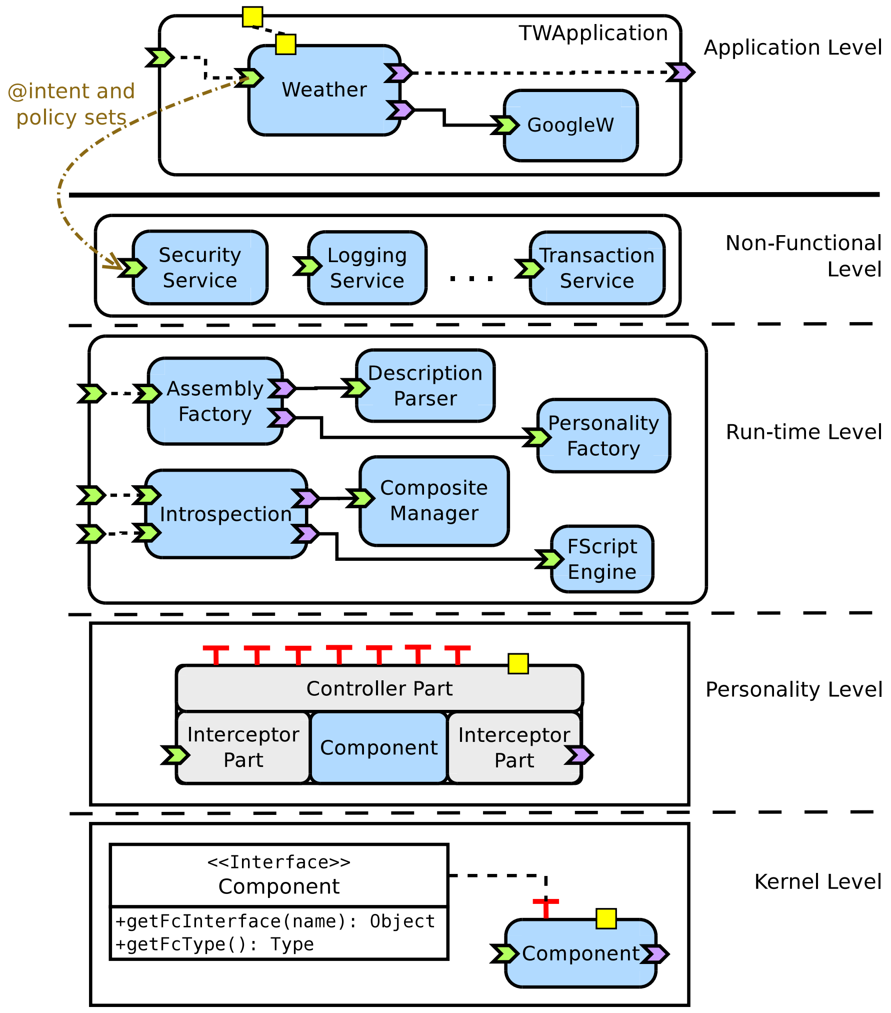


Figure 2.2: The FRASCATI SCA middleware stack (adapted from [Seinturier *et al.*, 2012]).

`DescriptionParser`, `PersonalityFactory`, and `AssemblyFactory` are the components in charge of the instantiation phase for executing SCA applications. The `DescriptionParser` loads and checks the user-defined file that specifies the SCA application composites (i.e., the composite descriptor file) and creates the respective run-time application structure. These descriptor files must conform to the SCA assembly specification. However, despite of being compliant to the SCA specification, the user can include in these files any of the FRASCATI extensions, that is, components and services with functionalities that are not part of the SCA specification, such as new binding types. The `AssemblyFactory` creates the component assemblies that correspond to the run-time model created by the `DescriptionParser`, including components, properties, implementations, services, interfaces, wires, and binding. Similarly, the `PersonalityFactory` adds the controller and interceptor parts to each component, according to the descriptor file specification, such as specific non-functional (e.g., authentication, logging) requirements.

On the other hand, the `CompositeManager`, as its name implies, manages the internal representation of the executed application as SCA elements, at run-time. It also provides the introspection services (through the `Introspection` component and services), not only for the executed applications, but also for all of the FRASCATI SCA implementation components and services. The `Introspection` component also provides access to the `FScriptEngine` component, the reconfiguration-primitives execution engine.

4. The Non-Functional Level. This level corresponds to the implementation and support of the SCA Policy Framework specification [Beisiegel *et al.*, 2007b]. As introduced previously, this specification provides basic support for fulfilling non-functional requirements in the executed applications. This support is realized through annotations in the composite descriptor files. For instance, the `@Confidentiality`, `@Integrity`, and `@Authentication` annotations can be used to implement confidentiality, integrity and authentication functions in service invocations, respectively.

FraSCAti implements the corresponding non-functional services using interception mechanisms, where each policy is associated with exactly one component (or composite).

2.2.4 FraSCAti vs. Other Implementations: SCA Challenges

Concerning the capabilities of introspection and primitive reconfiguration (the most relevant characteristics for the goals of this dissertation) and performance, FraSCAti has been evaluated positively over some of the most representative SCA implementations, such as Apache Tuscany[17], and Fabric3[18] [Seinturier *et al.*, 2009, Romero, 2011]. Other important SCA implementations, such as IBM's WebSphere Application Server V8 for SCA[19], does not implement dynamic reconfiguration capabilities yet, even though these characteristics are planned in the product road-map [Bentancour *et al.*, 2011].

However, even with the additional characteristics that FraSCAti offers over the SCA specification and other implementations, it still has the SCA limitations of policy sets for the autonomous and dynamic fulfillment of varying QoS-level objectives under system execution's changing conditions. In particular, [Papazoglou *et al.*, 2007] identified the need for dynamic reconfiguration capabilities in SCA and component platforms as one of the main challenges for service foundations in the Service Oriented Computing paradigm. [Seinturier *et al.*, 2012] presents a partial answer to this challenge, providing FraSCAti with capabilities for *primitive* reconfiguration. In this dissertation we address the challenge of dynamic reconfiguration at a larger scale, that is, to satisfy high-level objectives (e.g., QoS-level objectives) performed in an autonomous and reliable way.

2.3 Quality of Service (QoS) Software Contracts

Software systems must satisfy functional and extra-functional requirements. Quality of Service (QoS) requirements—such as those on performance, availability, confidentiality, and reliability—are classified among the latter. As their verification usually requires of information gathered from the system operation (i.e., at run-time), QoS requirements are also known as *operational* quality requirements [Beugnard *et al.*, 1999, Bosch, 2000].

In the last years, ubiquitous software services (e.g., Web, REST, RPC, RMI and other remotely available services) have gradually pervaded all aspects of everyday life. The subsequent proliferation and massive use of these services, individually or combined among them and with traditional ones (e.g., Web mashups and wrapped legacy applications), challenge the satisfaction of their QoS requirements at run-time. In effect, when confronted with their dependencies on the dynamic nature of context, new and highly dynamic requirements appear for these services. These new requirements, such as having to fulfill changing QoS levels under different

[17]https://cwiki.apache.org/TUSCANY/sca-java-2x-releases.html
[18]http://www.fabric3.org
[19]http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca

context situations, further exacerbate the problem of guaranteeing the expected quality of these services, especially under varying conditions of system execution. Additionally, these dynamic capabilities are also expected to be performed in a reliable way, in order to be trustworthy and acceptable by users.

QoS contracts constitute a natural and effective means for specifying these operational-quality and context-dependent requirements. These contracts are subject to re-negotiation, and thus, must be managed at runtime [Beugnard *et al.*, 1999, Krakowiak, 2009, Tran and Tsuji, 2009]. QoS contracts should define expected *levels of operation* (QoS levels) to be satisfied by a software application. These QoS levels should be specified for each of the different context conditions to be faced by the software application during its execution. Thus, the continuous satisfaction of a QoS contract (i.e., its preservation) implies to monitor each of these QoS levels at run-time, but also to restore their satisfaction (e.g., using guaranteeing actions) whenever they are violated (e.g., when the respective context conditions change).

Several specifications [Frølund and Koistinen, 1998, Keller and Ludwig, 2003], languages [Röttger and Zschaler, 2003, Becker, 2008], models [ISO, 2001, Collet *et al.*, 2005, Chang and Collet, 2007b, Lee *et al.*, 2009, Comuzzi and Pernici, 2009], and formal semantics [Braga *et al.*, 2009, Cansado *et al.*, 2010], among others, have been proposed to specify, model and support the characteristic attributes of QoS contracts, 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 as an open challenge, as we analyze in the next sections.

2.3.1 QoS Contract Specification

Most of the QoS contract specifications, models and languages consider the characteristics identified by Frølund and Koistinen in their proposed QML (QoS Modeling Language) [Frølund and Koistinen, 1998]. This language specifies QoS properties, and respective conditions on them, as requirements for individual software components to be addressed at design time. For this, QML establishes a fixed vocabulary associated to different QoS properties and allows to define contract types, similar to structured types of common programming languages. CQML, CQML+, as well as other variations of QML, also address QoS contracts specifications to be used at design time and identifies several additional characteristics to QML for specifying QoS contracts. One of such additions is the relationships among the QoS provisions (i.e., obligation and CPU-memory-network resource dependencies among contract satisfaction) [Röttger and Zschaler, 2003, Becker, 2008]. Nevertheless, these languages have some limitations. First, they have no means for specifying different QoS levels to be monitored and fulfilled at runtime for a given QoS property. Second, even though some of these approaches derive alternative specifications for the runtime representation of QoS properties, their semantics is not formally specified. Thus, they leave the problem of dynamic fulfillment of the QoS contracts for the runtime platform system.

On the services side, the Web Service Level Agreement (WSLA) specification [Keller and Ludwig, 2003] focuses on defining a syntactic structure for conditions expressing Service Level Agreements (SLAs). Data gathered from monitor probes in the managed service implementation is fed into this structure to be transformed, evaluated, and produce context events, if relevant. The WSLA also allows to define a guaranteeing action to be executed in response to events notifying the violation of SLAs [Ludwig *et al.*, 2003]. However, the semantics of these actions is defined informally, and limited to operations such as chained event notification, not addressing directly how to reconfigure the services to restore the contract satisfaction.

The model proposed by Jureta et al. integrates several previous quality models for software services and refines the specification of units and gathered data transformation. It also refines the modeling of dependency and priority between quality metrics [Jureta *et al.*, 2009]. Despite the syntax of the specification model is formalized, as in the case of the WSLA guaranteeing actions, the lack of a formal semantics limits the automated verification and reasoning on contracted quality properties.

Challenges Identified on Contract Specification

The main challenges identified with respect to QoS contract specification are the following:

- The expression of the QoS contract itself, given that it must specify (i) the different contextual conditions on the contracted QoS property; (ii) the corresponding guaranteeing actions to be performed in case of the QoS contract violation; and (iii) the responsibilities of the software elements intervening in the contract-preservation process. In particular, a QoS contract should specify the monitoring and guaranteeing elements for the contracted properties.
- Even though contract specification addresses only the contract definition, having a precise semantics is also necessary, not only for automating the monitoring and verification of the QoS levels to be satisfied, but also for reasoning on the process of maintaining the QoS properties fulfilled, thus enabling the possibility of restoring the system to a consistent state.

2.3.2 QoS Contract Management and Fulfillment

Regarding software contract management in general, several approaches have been proposed since the first ideas on functional contracts introduced by Floyd and Hoare [Floyd, 1967, Hoare, 1969]. In the object-oriented paradigm for software development, one of the most influential of those approaches is the *Design by Contract Theory* introduced in the Eiffel programming language [Meyer, 1992]. 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 functional contracts. By using assertions as integral parts of the source code to be verified at runtime, routines are made self-monitoring from compile-time. The violation of an assertion, such as a class invariant, is automatically managed by the standard *rescue* clause. In this clause, the programmer must handle appropriately the causes of the assertion violation in order to restore a program's consistent state.

At least in abstract, most of the approaches addressing the fulfillment of contracted QoS levels follow the *rescue* clause strategy implemented in Eiffel. As illustrated in the previous sections, QoS contracts must define expected QoS levels to be satisfied by a software application. Whenever any of these QoS levels are reported as (in risk of being) violated by the corresponding monitor, guaranteeing actions are applied to restore the contract satisfaction (or prevent its violation). In this sense, QoS contract satisfaction is, in general, a form of property preservation. For example, the approach of Delaval *et al.*, based on the Fractal component platform, focuses on the property of safety [Delaval and Rutten, 2010]. Inspired by control theory, their approach synthesizes static reconfiguration controllers from a contract specification. However, the reconfiguration transition function must be written by the user, for every transition on every state. Another approach, aiming at preserving system structural properties in software reconfiguration is the proposed by Hnětynka and Plášil in [Hnětynka and Plášil, 2006].Their approach limits the system reconfigurations to those matching three specific reconfiguration patterns that precisely model the conditions of dynamic reconfigurations that avoid the introduction of system architecture inconsistencies.

Chang *et al.* proposed *ConFract*, a contracting system for hierarchical components that supports self-healing and self-protection, by combining contracts with feedback loops [Chang *et al.*, 2006]. Their approach focuses on modeling contracts as runtime objects to support different negotiation policies in order to restore the validity of contracts by adapting components and the contracts themselves. Contract violations must be managed with *ad hoc* code written by the user. *ConFract* contracts are specified in an OCL-like language as preconditions, postconditions and invariants, whose scope is limited to components and interfaces.

Addressing the fulfillment of system-wide contracts on extra-functional properties, Chang *et al.* focus on the problem of combining low-level properties of individual components to obtain system-level properties to support contract negotiation [Chang and Collet, 2007a]. Their approach identifies compositional patterns for combining non-functional properties. However, their system-wide properties must be computable as a function of the properties of the software components involved. Thus, with this approach it is not possible to consider the dependencies and interactions neither among the components themselves of the managed application, nor among the system and the execution context, nor the dependencies on system usage.

Bucchiarone *et al.* and Ehrig *et al.* used graph transformations for analyzing and verifying specific self-healing properties at design time [Bucchiarone *et al.*, 2009, Ehrig *et al.*, 2010]. Their proposal use a fixed set of particular transformation rules to be applied in response to system failures. Thus, from the application of these rules the self-healing properties are derived and proved.

On the formal semantics side, Cansado *et al.* defined behavioral contracts based on a labeled transition system as a formalism to unify behavioral adaptation based on property composability and interface adaptability [Cansado *et al.*, 2010]. With this formalism, they focus on determining if a service re-composition can be performed, based on the possibilities to adapt the provided and required interfaces of a different provided and requires services, even if these interfaces are syntactically different.

Braga *et al.* formalized the semantics of a QoS contract language using operational calculus rules driven by a state machine [Braga *et al.*, 2009]. To guarantee the satisfaction of QoS constraints on the resulting states, they translate QoS specifications to the Maude model checking tool. The given formal semantics also considers actual monitored context conditions in its transitions, and its guaranteeing actions are based on changing the compromised service by another that address the cause of the violation. The proposed reconfiguration by this approach, as well as the analyzed previously, is nonetheless limited to interface re-wiring of services.

Fiadeiro and Lopes presented a formalization of dynamic reconfiguration of business process workflows in terms of service discovery, binding, and orchestration operations for the SOA domain [Fiadeiro and Lopes, 2010]. For each level of abstraction, they use different formalisms (linear-time logic, graphs, partial algebras and state machines, although the latter is not detailed) to model the structural and behavioral corresponding aspects. The formalization is then proposed as a semantic domain with a corresponding operational semantics that enables ADLs to be extended with dynamic reconfiguration operations. However, these approaches assume that their reconfiguration strategies can always find services to satisfy the contracts, independently of the occurrence of unexpected context situations.

Challenges Identified on Contract Management and Fulfillment

The main challenges identified from these approaches, with respect to QoS contract management and fulfillment, are the following:

- From the components perspective and following its foundations [Szyperski, 1998, Heineman and Councill, 2001], the strategies for addressing QoS contracts are traditionally bottom-up. That is, contracts as a whole rely on some function that evaluates the system QoS levels depending on explicitly identified responsibilities of single components [Collet *et al.*, 2005, Léger *et al.*, 2010, Delaval and Rutten, 2010]. However, as software services are composed of several components, their QoS properties do not result exclusively from the joint work of their constituting components, but also from their interactions with the underlying operating system and, more importantly, from their execution context. Hence, QoS contract preservation requires context-aware strategies that dynamically and reliably manage subsets of components and their interrelationships, as a whole, considering their differences between the software application states [Zeng *et al.*, 2004a, Yang *et al.*, 2009].
- Several strategies can be used as guaranteeing actions to address each expected QoS level on a QoS property. These strategies have been 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 [Barbacci *et al.*, 1995, Buschmann *et al.*, 2007].
- The treatment of QoS contracts have traditionally focused on what contracts must specify and accomplish, that is, on guaranteeing QoS obligations. Nonetheless, given the unpredictable and complex nature of context, and that QoS properties depend on it, QoS contracts are required to be managed robustly. Robustness is necessary to maintain the consistency between the contract states and the states of the software subject to the contract conditions over time, even if the software state deviates from the expected ones. In some sense, this is related to the importance of addressing also what contracts leave as unspecified, in order to face context uncertainty. For instance, a relevant question in this setting is: what should a component run-time system do, and in which state should it leave the executed software system when facing an unspecified context situation that is already disturbing the QoS contract satisfaction? We call this requirement *robustness with respect to context unpredictability*, as identified similarly in [Murray *et al.*, 2003, Goldsby and Cheng, 2008]

2.4 Self-Adaptive Software Systems

Self-adaptive software systems evaluate their own behavior at run-time and reconfigure themselves whenever they are no longer satisfying their requirements [Shaw, 1994, Oreizy *et al.*, 1999]. As illustrated in Fig. 2.3, the system requirements satisfaction (e.g., contracted QoS levels) can be disrupted at run-time by changing conditions of execution (i.e., context changes), such as operational environment variations, user needs modifications, and system intrusions or faults. To maintain its requirements satisfied, the system must be reconfigured, for instance, by augmenting or changing the system components and services, in order to continually optimize, protect, or recover itself [Cheng *et al.*, 2009a, Taylor *et al.*, 2009, Tamura *et al.*, 2011a].

Over the past century, the feedback-loop model defined in control theory has been used as a reference in multiple fields of engineering with substantial advances, for instance in the automation of industrial processes [Ogata, 1996]. Inspired by this model, IBM researchers defined the *autonomic element* as a building block for developing self-managing and self-adaptive software systems, in the form of the so-called Monitoring-Analysis-Planning-Execution and shared Knowledge (MAPE-K, or simply MAPE) loop. The purpose of this model is

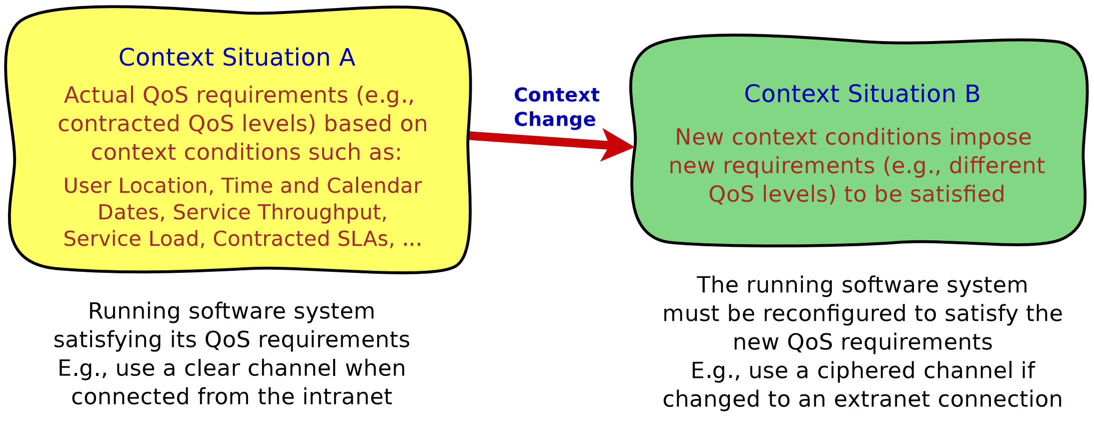

Context Situation A
Actual QoS requirements (e.g., contracted QoS levels) based on context conditions such as:
User Location, Time and Calendar Dates, Service Throughput, Service Load, Contracted SLAs, ...
Context Change
Context Situation B
New context conditions impose new requirements (e.g., different QoS levels) to be satisfied
Running software system satisfying its QoS requirements E.g., use a clear channel when connected from the intranet
The running software system must be reconfigured to satisfy the new QoS requirements E.g., use a ciphered channel if changed to an extranet connection

Figure 2.3: Changing context situations vs. different QoS requirements to satisfy.

to develop autonomous controlling mechanisms to regulate the satisfaction of dynamic requirements, specifically in software systems [Kephart and Chess, 2003, Hellerstein *et al.*, 2004, IBM Corporation, 2006].

In the two past Dagstuhl Seminars on Software Engineering for Self-Adaptive Systems[20], three challenging aspects were selected as critical for advancing and leveraging the engineering of self-adaptation in software systems: (i) the visibility of the feedback loop (followed as a reference model) in the system design; (ii) the management of context uncertainty; and (iii) assurances and properties [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012]. In the following sections we analyze the foundational principles of the engineering of self-adaptive software systems and their main challenges, as related to the goals of this dissertation.

2.4.1 Revisiting the MAPE-K and Feedback Loop Models

In Fig. 2.4 we illustrate our interpretation of the MAPE-K loop in terms of the feedback-loop block diagram. To autonomously satisfy the regulation of its requirements (cf. *reference control input*s in the figure), which vary with context changes, a `Monitor` gathers information from the internal and the external contexts. This information, in the form of *control symptoms*, is analyzed by the `Analyzer`, which compares them to the *reference control input*, yielding a *control error*. Based on this difference, the `Planner` element computes *control actions* to be instrumented by the `Executor` in the managed software system. A `Knowledge Manager` manages relevant information, such as adaptation policies, thresholds, and rules, shared by the other MAPE-K loop elements. The measured control data can also be affected by *context disturbances* caused, for instance, by the system adaptation itself [Hellerstein *et al.*, 2004, Villegas *et al.*, 2012].
The elements and functionalities of the MAPE-K loop, as applied for regulating the satisfaction of contracted QoS levels, are the following [Kephart and Chess, 2003, IBM Corporation, 2006]:

Monitor. Monitoring elements are responsible for sensing changes in both, the managed application's internal variables corresponding to QoS properties (i.e., *measured QoS data*), and also the *external context* (i.e., measured from outside the managed application). Monitors must notify relevant *context events* to the analyzer, based on these changes. Relevant context events are those related to the specified in the system requirements (e.g., in QoS contracts).

Analyzer. The analyzer, based on the high-level requirements to fulfill, and the context events notified by monitors, determines whether an adaptation must be triggered. This would occur,

[20]http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=10431

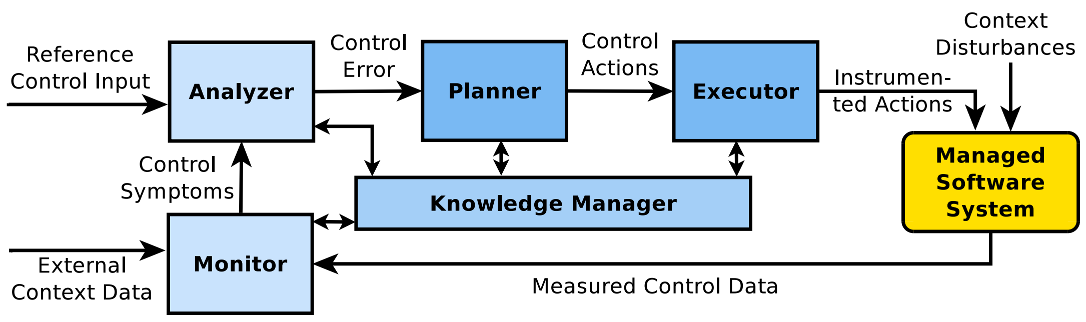


Figure 2.4: The feedback-loop interpreted in the MAPE-K loop block diagram.

for instance, when the events notify about changes that violate the expected QoS level (i.e. reference control inputs). Context analyzers can be based on either, multi-event or single-event pattern matchers, as discussed in [Luckham, 2001] and [Hermosillo *et al.*, 2010]. Multi-event matchers produce complex events based on single events that accumulate over time. These single events are produced by single-event matchers, which identify partial matches in the flow of the monitored events.

Planner. Once notified with a reconfiguration event from the context analyzer, the planner selects a strategy to fulfill the new requirements, using the shared Knowledge base. From the application of the selected strategy, it computes the necessary control actions to be instrumented in the managed software system. An important difference between the feedback and the MAPE loops is that, in the first, the control actions are continuous signals for physical actuators (e.g., resistors and motors), whereas in the second, they must be sequences of discrete operations (thus called reconfiguration plans). These discrete operations are then interpreted by the executor.

Executor. Upon reception of a reconfiguration plan, the executor interprets each of the operations specified in the plan and instruments them in the managed software system. This implies to translate or adequate the reconfiguration actions to the characteristics of the particular component runtime platform used to execute the managed software system.

(Reconfiguration) Knowledge manager. The reconfiguration knowledge manager makes explicit the relevant knowledge about the managed software application configuration, and how to perform its re-configuration at runtime. In a feedback loop, the adaptation controller encodes fundamental knowledge about the properties of the physical plant or target system to control in the so-called system transfer function. Based on this mathematical model of the target system response to context disturbances, adaptation controllers can be guaranteed on adaptation properties, such as the short settling-time, the stability, the accuracy, and the small resource-overshoot [Ogata, 1996, Hellerstein *et al.*, 2004]. In contrast to physical systems, built from materials with well known standard properties such as conductance, capacitance, and heat conduction, software systems are developed with software components with no standardized properties. Thus, in the case of the MAPE loop (i.e., in the software systems domain), the knowledge base, provided by the adaptation designer, must supply the lack of information about the properties of the managed software application, in order to make adequate decisions for its adaptation.

2.4.2 Other Models for Self-Adaptation in Software Systems

Since the introduction of the MAPE loop, several researchers have proposed other models applying the feedback-loop concept for engineering self-adaptive software (SAS) systems

[Salehie and Tahvildari, 2009]. In this section we analyze some of the most representative of them, in light of the visibility of feedback-loop elements as one of the main concerns identified by the research community on software engineering for self-adaptive and managing software systems (SEAMS) [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012].

Visibility of Feedback-Loop Elements

The importance of maintaining the feedback-loop elements (and the feedback-loop itself) as visible in the design of software systems for controlling adaptation processes has been identified in several research works.

For instance, Shaw presented a design model that emphasizes on decoupling the feedback-loop elements (i.e., comparison, plan correction, and effect correction), and identifies the importance of the execution context to guide adaptation processes [Shaw, 1994]. Similarly, Müller *et al.* and Giese *et al.* analyzed the benefits of specifying the feedback loops and their major components explicitly and independently, as well as the necessity of making explicit the interactions among the feedback-loop elements. The first also emphasizes on the importance of maintaining these elements explicit from analysis and design to implementation [Müller *et al.*, 2008], whereas the second argues for the management of the context complexity, and the interactions among multiple feedback loops when more than one is necessary [Giese *et al.*, 2009].

In the same way, the autonomic computing reference architecture (ACRA) makes the feedback loops in autonomic systems explicit [IBM Corporation, 2006]. The ACRA's reference architecture provides a guide to organize and orchestrate an autonomic system in a three-layer hierarchy of structured building blocks composed of autonomic managers (i.e., MAPE loops), knowledge sources and manageability endpoints (management interfaces), as illustrated in Fig. 2.5.

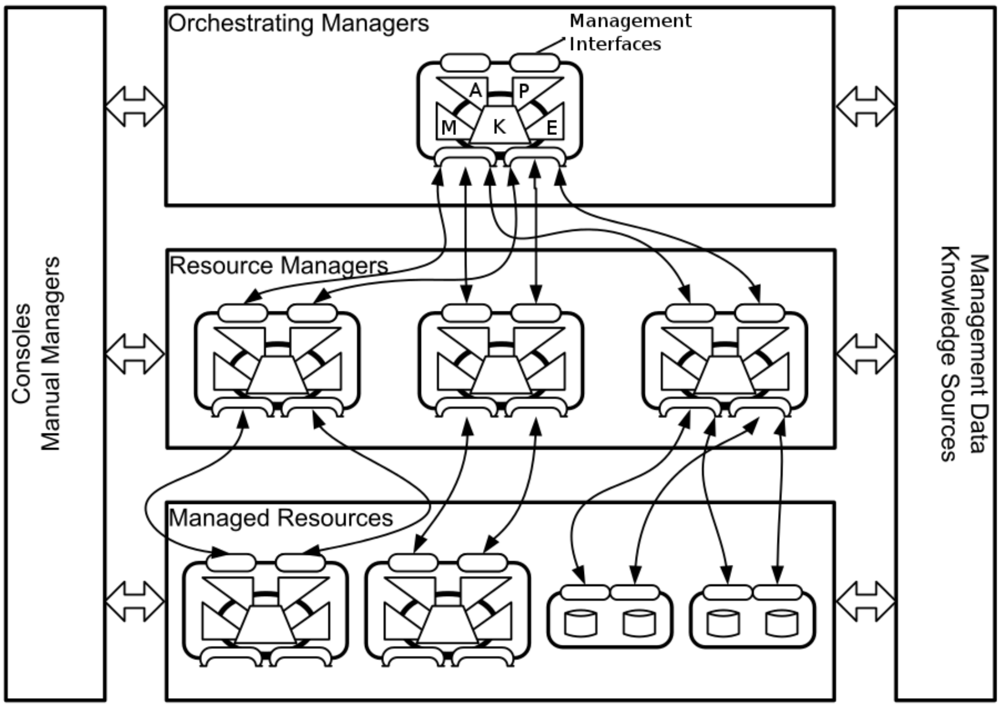


Figure 2.5: The autonomic computing reference architecture (ACRA, [IBM Corporation, 2006]).

Kramer and Magee proposed a three-layer reference architecture for self-managed systems.

These layers correspond to goal management, change management, and component management, with independent time scales of control [Kramer and Magee, 2007]. Based on Gat's three-layer architecture [Gat, 1998], the component layer reconfigures software components and ensures application consistency. The change management layer, handling decentralized configuration management, copes with inconsistent views of the system state with the goal of re-establishing a satisfactory stable state. Finally, the goal management layer addresses planning to achieve high-level goals. Similarly, Litoiu *et al.* define three levels of autonomic management for provisioning, application tuning, and component tuning [Litoiu *et al.*, 2005].

Yet other schemes for controlling adaptation of software systems is organizing multiple control loops in the form of a hierarchy, and even distributing the MAPE-loop elements in different machines. Examples of these decentralization variants and extensions are the proposed by Vromant *et al.* and Weyns *et al.* Their decentralized architectural models combine the MAPE-loop elements in all possible ways, providing guidance to design decentralized adaptation controllers whose elements are physically distributed [Vromant *et al.*, 2011, Weyns *et al.*, 2010].

Nonetheless, despite ACRA and the MAPE loop helps to improve the visibility of feedback loops, the internal components of each control-loop and the control-loop itself still remain hidden inside the autonomic manager. Certainly, the specification of the autonomic manager provided in the IBM architectural blueprint for autonomic computing characterizes the manager as *a component that implements an intelligent control loop* [IBM Corporation, 2006]. Moreover, even when the ACRA architecture drivers are clearly the feedback loops in the form of autonomic managers, their internal elements (i.e., the elements of the MAPE loop) are highly coupled. Therefore, even though the multiple feedback loops defined in an ACRA-based model can be distributed for instance to improve the system scalability, this distribution is limited by the autonomic manager boundaries. Furthermore, Müller *et al.* as well as Kramer and Magee analyze that even though feedback loops have been recognized as fundamental design elements for self-adaptation, their visibility is usually hidden in the related approaches. In many cases, the self-adaptation mechanisms are intertwined with the managed applications, rendering them as hard to reuse and manipulate, and more importantly, as unanalyzable and non-comparable in their inherent properties [Kramer and Magee, 2007, Müller *et al.*, 2008, Müller *et al.*, 2009, Villegas *et al.*, 2011b].

2.4.3 Particular Approaches for Self-Adaptation

A first example of concrete implementations is Rainbow, the adaptive framework for implementing self-healing software systems developed by Garlan *et al.* [Garlan *et al.*, 2003]. This project analyzed the use of software architectural models at runtime as the basis for reflection and dynamic adaptation, and its architecture maps directly to the feedback control architecture proposed by Shaw [Shaw, 1994, Müller *et al.*, 2008].

Solomon *et al.* proposed a real-time adaptive control approach for autonomic computing environments [Solomon *et al.*, 2007]. Their adaptive control is based on a multi-layer architecture similar to ACRA, where the two upper layers correspond to the autonomic system adaptation and the autonomic system layers respectively, and the lowest layer corresponds to the managed infrastructure. The autonomic system adaptation layer adapts the autonomic system layer whenever the management objectives are not achieved.

In the self-organizing systems community, Caprarescu and Petcu proposed a decentralized autonomic manager composed of many independent lightweight feedback loops implemented as agents, where each agent is an implementation of a MAPE loop [Caprarescu and Petcu, 2009]. Control objectives in this approach are specified as policies, where each feedback loop agent uses just one policy that is shared among all the agents organized in the same group. At the architectural level, this approach is based on the three layers proposed by Kramer and Magee [Kramer and Magee, 2007].

Self-Adaptation Assurances and Properties

Self-adaptation has been proposed as a strategy to maintain or improve the continued satisfaction of functional and extra-functional requirements under changing conditions of system execution. With this general purpose, it has been used to achieve different higher-level self-* goals, such as self-healing, self-recovering, self-protecting, and self-managing. For instance, [Candea *et al.*, 2004, Sicard *et al.*, 2008, Cardellini *et al.*, 2009] (and in some extent [Garlan *et al.*, 2004, Cheng *et al.*, 2009b]) address availability and reliability through self-healing and self-repairing strategies. In these approaches, based on the dependency between availability and the mean time to recover (MTTR) from system failures, the MTTR is used to evaluate the continuity of agreed service (i.e., the reliability). By obtaining experimental measurements of the MTTR under different scenarios for a given application, they offer the respective results as a guarantee for the (average) response time that the adaptation mechanism takes to recover from a failure.

Other strategies used to guarantee reliability are the proposed by Léger *et al.* and Delaval *et al.* Based on the Fractal component platform [Bruneton *et al.*, 2006][21], the first defines reliability for system reconfiguration as the preservation of the structural constraints defined in the Fractal component model, while preserving system availability [Léger *et al.*, 2010]. To guarantee reliable reconfigurations, they extended the Fractal's FPath/FScript navigation and reflection textual languages with transactional (Atomicity, structural Consistency, Isolation and Durability –ACID) properties. The second work addresses *safety*. Inspired by control theory, their approach synthesizes static reconfiguration controllers, correct by construction. This synthesis is performed from a contract specification given by the user, which determine all possible application states. For each of the transitions from every state to the others, the user must write a function for computing the application reconfiguration [Delaval and Rutten, 2010].

Nonetheless, despite the significant advances achieved in self-adaptation, guaranteeing the satisfaction of self-* goals requires of standardized methods and properties to verify their accomplishment [Salehie and Tahvildari, 2009]. More importantly, self-adaptation assurances and properties have been identified as one of the most challenging current barriers for the wide adoption of self-adaptive software, and leveraging the benefits of the engineering of this kind of systems [Werner Dahm, 2010, Villegas *et al.*, 2011b, de Lemos *et al.*, 2012].

Management of Context Uncertainty

Self-adaptive software systems are continuously confronted with context changes, which are, naturally, the main reasons for their adaptation. As a result, this kind of systems must be prepared to manage unexpected changes that occur in their execution context.

To cope with these unexpected changes, Murray *et al.* analyzed the use of feedback loops as an explicit strategy to address robustness with respect to context uncertainty [Murray *et al.*, 2003]. In addition, Cheng *et al.* and de Lemos *et al.* identified context uncertainty as one of the most challenging problems faced by context-aware software systems, given its complex and dynamic nature [Cheng *et al.*, 2009a, de Lemos *et al.*, 2012]. In this respect, the approach by Goldsby and Cheng models dynamically adaptive systems (DAS) as state machines, with transitions as system reconfigurations [Goldsby and Cheng, 2008]. Inspired by the adaptability of living organisms, they model systems using UML diagrams that satisfy functional and

[21]http://fractal.ow2.org

non-functional invariants. Based on these diagrams, and by applying digital evolution techniques, they dynamically generate not only one, but several target states for a given transition, and then assist the user to select the target system with the most appropriate QoS provisions. Thus, they address context uncertainty by generating several possible target systems with qualitatively different QoS characteristics, which satisfy the user specified QoS invariants, and leaving the crucial decision of what to do to the user.

Lin *et al.* analyzed the intrinsic uncertain nature of context as the practical impossibility of modeling it completely, accurately, and consistently, even in short intervals of time [Lin *et al.*, 2009]. Thus, to avoid human intervention, which implies long interruptions on the system execution because of the difficulty in deciding which action to choose as the next step, they refine context management to specific context situations. More specifically, they use Bayesian networks, learning techniques, and monitoring refinement to address the context information incompleteness problem. In this way, their approach fill-in missing context information with aggregated values such as mean values and statistical values on historical data.

Given that the environment may change in unexpected ways, the system may adapt in such a way that was not foreseeable at design or configuration time. However, most of the self-adaptive proposals assume that their approaches can always cope well with the context changes (unexpected or not), and produce the desired results. In this respect, as identified by Cheng *et-al.* Müller *et-al.* and de Lemos *et-al.*, the main challenge for adaptation mechanisms is to manage context uncertainty, guaranteeing not only that the adaptation process and its results are reliable, but also that it can cope with context situations unforeseen by the system evolution designer [Cheng *et al.*, 2009a, Müller *et al.*, 2009, de Lemos *et al.*, 2012].

2.5 Example Application Scenario: A Reliable Videoconference System

To better understand the requirements for dynamic and self-reconfiguration implied by the continued satisfaction of QoS contracts, we use a simplified version of a reliable mobile videoconference system (RVCS) as application example.

The RVCS system manages corporate videoconference meetings and provides services to users for registering and attending virtually to these meetings. A meeting is conducted by a presenter, who exposes a given matter to a group of attendants. The presentation audio and video is transmitted to the virtual attendants, which can ask questions via chat. The quality of these services are subject to a QoS contract that guarantees expected QoS level obligations on two QoS properties, namely, confidentiality and availability. Table 2.1 illustrates, for these two properties, the expected QoS levels under the possible context conditions the application can confront in its execution. Thus, software clients are expected to be responsible for maintaining the services to the user in a "smart" way, satisfying the requirements illustrated in Fig. 2.6.

Table 2.1: QoS contractual conditions and corresponding Service-Level Objectives for:

(a) Confidentiality (based on corporate network access)

Context Condition	Service Level Objective
C1: Intranet Connection	Clear Channel
C2: Extranet Connection	Confident Channel
C3: No Netw. Connection	Local Cache

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

Context Condition	Service Level Objective
C4: $Bandwidth \leq 12$	Call on Hold
C5: $12 < Bandwidth \leq 128$	Voice Call
C6: $128 < Bandwidth$	Voice&Video Call

As confidentiality is a concern for this system, connections from the intranet are considered secure, thus clear communication channels can be used. From the extranet, confidential channels are required to be configured automatically, whereas in case of no connection, a local cache structure should be used. If the user goes into a low-bandwidth area, the application must reconfigure itself to drop the bi-directional video signals. As illustrated in the figure, a *QoS Reliability Management* element is needed to assume the responsibility of reconfiguring the application structure to address the QoS contract violation in each case (taking into account the application's actual state) in a transparent way.

Note that addressing these requirements statically (e.g., with *if-then* clauses) would not be satisfactory. First, the videoconference transmission and reception requires to be handled in different machines (i.e., server and client for each virtual attendant), being their respective contexts not necessarily the same. This difference would introduce synchronization and decision issues between the code in the two machines. Second, new QoS levels introduced as a result of contract re-negotiation could not be managed.

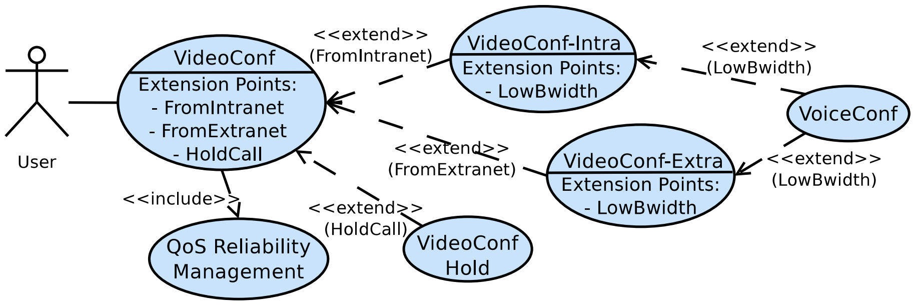


Figure 2.6: Use case diagram for the requirements of the RVCS example.

In this example, availability and confidentiality are QoS properties interpreted following [Barbacci *et al.*, 1995]. That is, the software application must ensure (i) the continued service of active videoconferences; and (ii) the confidentiality on videoconference transmissions, specially under changing conditions of the application execution. On these two QoS properties, the contractual interest is on establishing the minimum levels for service acceptability (*QoS-level objectives*), under the possible (foreseen) context conditions of execution.

Initially, assume the mobile user joins a video conference from her office at the corporate building through an intranet WiFi access-point. In this state, as the contractual condition $C1$ in Table 2.1a requires a clear-channel communication configuration, the application is expected to configure itself to satisfy that condition. A second application state is reached when the user 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 change of context, notified by a context event condition, signals an imminent violation of the confidentiality contract that was being fulfilled by the actual application configuration. In this situation, according to condition $C2$, a confidential-channel configuration on the mobile is required. The expected application behavior is then to reconfigure itself in response to this change, in a transparent way. It could, for instance, adopt one of the strategies for secure multimedia transport like those defined by [Ramachandran, 2002] or [Zeng *et al.*, 2004b], thus preserving the contract. The corresponding contrary reconfiguration would apply whenever the user 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 one. 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. Similar scenarios would occur for each of the videoconference virtual attendants.

2.6 Chapter Summary

In this chapter we have presented the concepts and foundations of the engineering of component-based software systems, QoS software contracts, and self-adaptive software (SAS) systems. As the three research areas involved in the development of this dissertation, we have also analyzed their current main concerns and key research challenges. For this, we have explored some of the models, approaches, and associated technologies that have been proposed to address the analyzed concerns, as well as papers that not only survey the respective states of the art, but also discuss their challenges for the near future.

We summarize the most important identified challenges, which correspond to the challenges addressed by this dissertation at different levels, as follows:

- In component-based software engineering:
  i. The preservation of QoS contracts, defined as the autonomous and dynamic fulfillment of their QoS-level objectives under changing conditions of system execution.
  ii. The reliable and high-level dynamic reconfiguration capabilities to satisfy high-level objectives (e.g., QoS-level objectives).
- In QoS software contracts:
  i. The precise definition (i.e., syntax and semantics) of QoS contracts to specify the different QoS levels to fulfill, context conditions, and guaranteeing actions, as well as the distribution of responsibilities among the elements intervening in the contract-preservation process.
  ii. The development or use of context-aware strategies, able to be parameterized, to manage subsets of components and their interrelationships to address specific QoS properties.
  iii. The robust management of contract states and the states of the software application subject to the contract conditions, with respect to context unpredictability.
- In self-adaptive software systems:
  i. The separation of concerns between the managed software application, and the adaptation mechanism. This is another way of expressing the necessity of maintaining the feedback loops and their internal components explicit in the design of self-adaptive software systems. Besides obtaining the benefit of reuse, the separation of concerns allows the adaptation mechanisms to be analyzable and comparable in their inherent properties.
  ii. The identification and definition of standardized properties and assessment methods to verify and guarantee the satisfaction of self-adaptation goals.
  iii. The management of context uncertainty, guaranteeing not only that the adaptation process and its results are reliable, but also that it can cope with context situations unforeseen by the user.

In the next chapter, we address the challenge of identifying and defining standardized properties inherent to self-adaptive software systems. These properties are identified from a framework that we define to characterize and evaluate adaptation mechanisms. The definition of these properties, which is our first main contribution in this dissertation, is fundamental to better understand our model for preserving QoS contracts through reliable dynamic reconfiguration, our second main contribution. The development of this model addresses the other identified challenges.

Part II

Contribution

Chapter 3

Quality-Driven Self-Adaptation Properties

Contents

In this dissertation we address two main goals. The first is to define *adaptation properties*, that is, properties that we identify as inherent to self-adaptive software (SAS), as a common basis to evaluate this kind of systems. The second is to preserve QoS contracts in component-based software, using dynamic reconfiguration with formal guarantees on the *reliability* of this reconfiguration. Even though the first goal has possibly a wider scope than the second in a general context, the SAS properties defined in this chapter are most significant in this dissertation for supporting the reliability guarantees pursued by the second goal. Thus, in the following chapters we analyze and express these reliability guarantees in terms of a subset of the properties defined in this chapter. The properties in this subset are selected such that they coherently and adequately fulfill the enunciated reliability guarantees.

The motivation for identifying and defining the SAS properties is originated from the extensive analysis of research publications on self-adaptive software performed along the development of this dissertation. From this analysis we could determine that adaptation properties and corresponding metrics are usually not identified nor explicitly addressed as such [Salehie and Tahvildari, 2009, Cheng *et al.*, 2009b, Grassi *et al.*, 2009, Andersson *et al.*, 2009, Kaddoum *et al.*, 2010, Villegas *et al.*, 2011b]. What is more important, even when most of these works were based on the same model, the MAPE loop, they were evaluated based on non-comparable and non-standard basis. Although these evaluations are independently valuable by themselves, they do not offer enough support to further improve neither each of the particular approaches with respect to others, nor the engineering of SAS systems, on common criteria. Furthermore, without this common criteria, it is practically impossible to combine these approaches to face the increasingly wider and more complex requirements on self-adaptation.

In this chapter we present our contribution to help solve this problem, which is a framework to classify and compare SAS systems. Our framework comprises a set of characterizing dimensions, a list of adaptation properties, and respective mappings from these properties to quality attributes. We develop this framework based on an analysis of the MAPE-K model—a software reconstruction of the feedback loop reference model from control theory—for two reasons: (i) the MAPE-K model is the most commonly used model to realize adaptation mechanisms for software systems; and (ii) feedback loops (the MAPE-K model precursors) and SAS systems share their essential nature. Both use specific mechanisms to instrument a given target system (or, in the SAS domain, a *managed software application*), aimed at achieving and controlling desired states on it. The required instrumentation is performed by applying computed actions (or transducing signals) based on information gathered from the target system itself.

To develop our framework, in Section 3.1 we first recall the properties used in control theory to assess feedback loops, and the difficulties for applying them to evaluate self-adaptive software systems. In this analysis we also consider the evaluation methods used in software engineering with the same purpose. In Section 3.2 we specify a set of dimensions that characterize self-adaptive software. These dimensions are useful not only to identify the inherent properties of SAS systems –our *adaptation properties*, but also for clarifying their definitions, given that these properties result from the relationships among these dimensions. In Section 3.3 we identify and define our proposed adaptation properties and analyze their relationships to common adaptation goals and quality attributes that have been used in representative works in this research area. In Section 3.4 we define our framework to classify and compare self-adaptive software systems. We also illustrate its application by using it to classify some of the aforementioned representative works on software self-adaptation.

Finally, it is worth noting that instead of addressing the dissertation problem directly, in this chapter we focus on an intermediate level between the stated problem and its solution space. This intermediate level helps to qualify acceptable solutions for the general problem of software self-adaptation. In this sense, in this dissertation our evaluation framework also proves its usefulness offering us the possibility of selecting a coherent subset of the defined adaptation properties to guarantee the reliability of the required solution.

Correspondences in this Chapter: *Addressed Challenge(s):* C1 –Adaptation mechanisms should be evaluated and improved through comparable and well-defined standard properties. *Goal(s):* G1 –Characterize key properties inherent to software self-adaptation as a common basis to evaluate self-adaptation mechanisms. *General contribution(s):* GC.PC –Properties inherent to self-adaptive software systems characterized and proposed for standardization. *Specific contribution(s):* PC.1 –SAS inherent properties identified and defined.

3.1 Feedback vs. MAPE-K Loops: Evaluation Differences and Difficulties

We started the preliminary phase of our analysis on assessment of SAS systems with over 80 research papers published during the past decade. From this set, more than one half were filtered-out mainly because either they presented very generic proposals with non-measurable properties for their evaluation, or they did not include enough information in the papers for our characterization purposes. From the analysis of the remaining papers, it is still worth noting the difficulty to identify properties that can be used to evaluate comparatively common characteristics of different self-adaptive approaches. Nonetheless, several important advances have been made based on feedback loops from control theory, the MAPE loop, and the recognized importance of quality attributes as a basis for understanding and improving adaptive processes. We summarize the most relevant differences between the evaluation of feedback loops and MAPE loops, and the corresponding difficulties to apply the properties in the first to evaluate instances of the second, as follows.

First, as we mentioned before, even though control theory has standard properties to evaluate a controller (i.e., short Settling-time, Accuracy, Stability and small resource Overshoot –the so called *SASO properties* [Hellerstein *et al.*, 2004]), these properties present several difficulties when applied to SAS systems. One difficulty is the difference on the nature of the elements used in control theory vs. those used in self-adaptive software. In control theory, these elements are physical, independent, and self-responding entities with mechanical and chemical properties; whereas in SAS, they are CPU-consuming, logical entities, with no standard properties defined. This physical dimension even establishes a clear and unsurpassable boundary between the controller and the system to be controlled (target system) in control theory. In the case of SAS systems, this corresponds to the limits between the adaptation mechanism and the managed software application, which can be even indistinguishable, as evidenced in many of the analyzed approaches. A second difficulty is that for a wide variety of systems, feedback loop controllers can be built with predictive models for the target system's behavior. To achieve this, continuous mathematics is used to characterize the target system response to known inputs. Software systems in general are, in contrast, nonlinear systems with multiple discrete variables whose behavior results very difficult to model in the same way [Hellerstein *et al.*, 2009]. A third difficulty is that SAS systems require additional properties for their assessment, given the discrete nature of software. For instance, in contrast to the continuous signals computed to control the temperature of a plant, a sequence of discrete operations must be produced to modify the architecture of a managed software application. For this particular case, properties such as atomicity and termination of the adaptation process (including the sequence of reconfiguration operations) become necessary and critical. Therefore, the properties used to evaluate feedback loops must be reinterpreted to be used for self-adaptive software, being this a non-trivial task. These difficulties exacerbate the already mentioned problem regarding the lack of explicitness targeting specific adaptation properties, and solving them effectively could take the next two decades, according to [Werner Dahm, 2010].

Second, the lack of explicitness for addressing adaptation properties as a goal to be measured in the analyzed approaches results in a lack of evaluation methods and metrics for these properties and for the adaptation mechanisms themselves. For instance, even though most of the analyzed proposals are motivated by specific adaptation goals (e.g., self-healing, self-configuring, self-protection), they focus only on the mechanism of self-adaptation, not measuring explicitly their properties. Some of these proposals measure the achievement of adaptation goals by evaluating diverse metrics, but not through comparable adaptation properties. This situation could be reversed by designing mechanisms for self-adaptation with measurable properties explicitly specified from the beginning. For some verifiable properties, this can be obtained by developing or using formal models as a basis for self-adaptation mechanisms, such as the one we present in this dissertation.

Third, having neither properties, nor evaluation methods precisely defined makes it very difficult to compare, reason and improve on the particular achievements of the engineering of self-adaptive software. For instance, from our analysis it was impossible to identify any measurable relationship between the adaptation goals and the evaluation of the adaptation strategies, on common comparable basis.

In light of these differences and difficulties, and consistently with the stated goals of this dissertation, in the following sections we limit ourselves to define a set of properties that we identify as inherent to SAS systems. The definition of these properties is based on the identification of common characteristics found in the aforementioned research papers. We re-synthesize these common characteristics as characterizing dimensions for SAS systems. From these dimensions and their relationships we extract the adaptation properties, which constitute the foundation for our framework for classifying SAS systems. Additionally, these properties should provide a common basis to evaluate adaptation mechanisms for this kind of systems.

3.2 Characterizing Dimensions for Self-Adaptive Software

In this section we propose six dimensions to characterize self-adaptive software (SAS) systems. For each of the characterizing dimensions, which we identified from the analysis of the feedback-loop block diagram presented in Fig. 3.1, we consider a list of standardized classification options. These options resulted from the combination of attributes used in control theory, as well as others proposed by recognized authoritative sources (e.g., the SEI), and from the analyzed papers. We also consider factors that affect them and metrics for their evaluation.

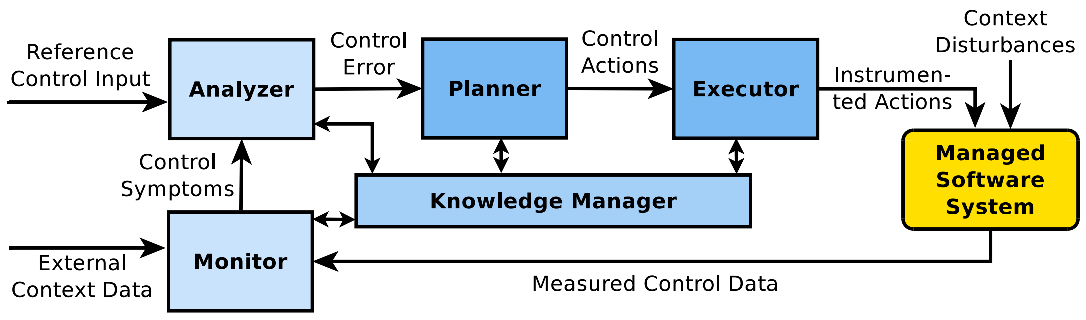


Figure 3.1: The characterizing dimensions for SAS systems identified in the feedback-loop diagram.

Adaptation goals. Although not explicit in the figure, these are the main reasons for the system or approach to be self-adaptive. Adaptation goals are usually defined through (a) one or more of the self-* (e.g., self-configuring, self-healing, self-protecting, self-optimizing and self-managing) goals; (b) the preservation of specific quality of service (QoS) properties; and (c) the regulation of functional or non-functional requirements.

Reference control input. The concrete and specific set of values and corresponding types that are used to specify the state to be achieved and maintained in the managed application by the adaptation mechanism, under changing conditions of system execution. Reference inputs are specified as (a) single reference values (e.g., a physically or logically measurable attribute); (b) some form of contract (e.g., quality of service (QoS), service level agreements (SLA), or service level objectives (SLO); (c) goal-policy-actions; (d) constraints defining computational states (according to the particular proposed definition of *state*); (e) expected values of utility functions; and even (f) functional requirements (e.g., logical expressions as invariants or assertions, regular expressions).

Measured control data. The set of values and corresponding types that are measured in the managed application. Naturally, as these measurements must be compared to the reference inputs to evaluate whether the desired state has been achieved, it should be possible to find relationships between these inputs and outputs. Furthermore, we consider two aspects on the measured outputs: how they are specified and how monitored. For the specification, the identified options are: (a) continuous domains for single variables or signals; (b) logical expressions or conditions for contract states; and (c) conditions expressing states of system malfunction. For monitoring, the options are (a) measurements on physical properties from physical devices (e.g., CPU temperature); (b) measurements on logical properties of computational elements (e.g., request processing time in software or CPU load in hardware); and (c) measurements on external context conditions (e.g., user localization or weather conditions).

Control actions. These are characterized by the nature of the required actions to control or modify the managed application, determining the output to be produced by the adaptation planner. This output is applied to (or instrumented in) the managed application to have an expected effect on it. The nature of these outputs is related to the extent of the intrusiveness of the adaptation mechanism with respect to the managed application. It also defines the extent in which the adaptation mechanism exploits the knowledge about either the structure or the behavior of the managed application in the adaptation process. Finally, these control actions can be coded by hand at design-time or synthesized dynamically at run-time. The computed control actions can be (a) continuous signals that affect the behavior of the managed application; (b) discrete operations affecting the computing infrastructure executing the managed application (e.g., host system's buffer allocation and re-sizing operations; modification of process scheduling in the CPU); (c) discrete operations that affect the processes of the managed application directly (e.g., processes-level service invocation, process execution operations—start/halt/resume, sleep/respawn/priority modification of processes); and (d) discrete operations affecting the managed system software architecture (e.g., software architecture reconfiguration operations).

System structure. Self-adaptive systems have two well-defined subsystems (although possibly indistinguishable): (i) the adaptation mechanism (or controller) and (ii) the managed application. One reason for analyzing adaptation mechanism and managed application structures is to identify whether a given approach implements the adaptation mechanism embedded within the managed application. Another reason is to identify the effect that the separation of concerns in these two subsystems has in the achievement of the adaptation goal. The analyzed approaches can be grouped into two sets: (i) those modeling the structure of the managed application to influence its behavior by modifying its structure; and (ii) those modeling the managed application behavior to influence it directly. The identified options for the adaptation mechanism structure are variations of the MAPE-K loop with either behavioral or structural models of the managed application: (a) single feedback control, that is, a MAPE-K structure with a fixed adaptation mechanism (e.g., a fixed set of transfer functions as a behavior model of the managed application) [Hellerstein *et al.*, 2004]; (b) adaptive control: a MAPE structure extended with managed application reference or identification models of behavior (e.g., tunable parameters of controller for adaptive controllers: model reference adaptive control (MRAC) or model identification adaptive control (MIAC)) [Narendra and Balakrishnan, 1997, Dumont and Huzmezan, 2002]; and (c) reconfigurable control: MAPE-K structure with modifiable algorithm for the adaptation mechanism (e.g., rule-based parameterized reconfiguration mechanism). For the managed application structure, the identified options are: (a) non-modifiable structure (e.g., monolithic system); and (b) modifiable structure with/without reflection capabilities (e.g., reconfigurable software components architecture). It is worth noting that not all options for the system structure can be combined with any options for the computed control actions. For instance, discrete operations affecting the computing infrastructure executing the managed application could be used to improve the performance of a monolithic system, whereas discrete operations for affecting this kind of software architecture in the managed application would not make any sense.

Adaptation properties. By adaptation property we mean a characteristic that is *inherent* to adaptation mechanisms of self-adaptive systems. That is, an adaptation property can be an specific attribute of the adaptation mechanism whose values satisfy given constraints, or a characteristic response to a known stimulus in a given context. Adaptation properties can be verified or measured in any adaptation mechanism. However, it is worth emphasizing the difference between adaptation properties and properties of managed applications. While the first apply to all adaptation mechanisms in general, the latter correspond to particular properties derived from the explicit requirements given by the user for a specific managed application, such as specific QoS properties. For instance, the *settling time* (i.e., the mean-time to perform the adaptation) can be measured in all adaptation mechanisms; however, guaranteeing the *confidentiality* in all communications from the extranet is a particular property that applies only to a specific managed application by effect of a requirement given by the user. Nonetheless, it is worth noting that despite in this dissertation we address the preservation of contracted QoS properties in managed software applications, our strategy to guarantee this preservation is through self-reconfiguration. Thus, we also must address the enforcement of adaptation properties.

The identified adaptation properties (i.e., the *inherent properties* for adaptation mechanisms) are (a) stability; (b) accuracy; (c) short settling-time; (d) small resources overshoot; (e) robustness (with respect to context unpredictability); (f) termination; (g) atomicity; (h) consistency (in the overall system structure and behavior); (i) scalability; and (j) security. We present our consolidated definitions for these properties in the next section.

3.3 Measuring Adaptation Properties

We classify the adaptation properties identified in the previous section according to *how* and *where* they are observed[22], as illustrated in Table 3.1. Concerning how they are observed, some of these properties can be measured or verified using static verification techniques while others require dynamic verification and run-time monitoring. With respect to where, these properties can be measured or verified on the managed application or on the adaptation mechanism. On one hand, some properties to evaluate the adaptation mechanism are observable on the adaptation mechanism itself, or on both the adaptation mechanism and the managed application; however, most properties can only be observed on the managed application. On the other hand, the user-defined QoS properties for the managed application are observable only on the managed application. In both cases, the environment (context) that can affect the behavior of the adaptation mechanism and the managed application is a factor worth of consideration.

Based on these observations, our model for evaluating self-adaptive software systems, despite focusing on the inherent properties of adaptation mechanisms, must also consider, in some cases, the evaluation of quality attributes on the managed application.

3.3.1 Adaptation Properties Inherent to Self-Adaptive Software

In this section we present our consolidated definitions for the properties that we identified as inherent to SAS systems in the previous section. The first four, called the *SASO properties*, correspond to our reinterpretations of the properties used in control theory to evaluate feedback loops [Hellerstein *et al.*, 2004]. The remaining properties were synthesized from the analyzed papers. Citations included in each property definition refer either to papers where the property

[22]A property is *observable* in a given point or element of a system (i.e., the *adaptation mechanism* or the *managed application*), with respect to *where* the property is measurable or verifiable, independent of whose element the property is.

Table 3.1: Classification of adaptation properties according to how and where they are observed

Adaptation Property	Property Verification Mechanism	Where the Property is Observed
Stability	Dynamic	Managed Application
Accuracy	Dynamic	Managed Application
Short Settling Time	Dynamic	Both
Small Overshoot	Dynamic	Managed Application
Robustness	Dynamic	Adaptation Mechanism
Termination	Static	Adaptation Mechanism
Atomicity	Static	Adaptation Mechanism
Consistency	Both	Managed Application
Scalability	Dynamic	Both
Security	Dynamic	Both

was defined or to examples of adaptive systems where the property is observed or measured in the adaptation process.

Stability. The degree in that the adaptation process will converge toward the control objective (i.e., the reference control input). An unstable adaptation will indefinitely repeat the adaptation (controlling) action with the risk of not improving or even degrade the managed application to unacceptable or dangerous levels. In a stable system, responses to a bounded input are bounded to a desirable range [Lu *et al.*, 2000, Meng, 2000, Appleby *et al.*, 2001, Parekh *et al.*, 2002, Floch *et al.*, 2006].

Accuracy. This property is essential to ensure that adaptation goals are met precisely, within given tolerances. Accuracy must be measured in terms of how close the managed application approximates to the desired state (i.e., to the reference input values) [Cardellini *et al.*, 2009, Solomon *et al.*, 2010].

Short settling time. The time required for the adaptive system to achieve the desired state. The settling time represents how fast the system adapts or reaches the desired state, being it impossible to define it in absolute terms for all application domains. Long settling times can cause the system to reach unstable states. Thus, despite the acceptability of its measurement depends on the application domain, it must be short enough to avoid other undesirable effects. This property is commonly referred to as reaction time, and also Mean Time to Repair (MTTR), the most critical factor determining availability and reliability. [Lu *et al.*, 2000, Meng, 2000, Candea *et al.*, 2004, Hellerstein *et al.*, 2004, Kumar *et al.*, 2007].

Small resources overshoot. The utilization of computational resources during the adaptation process to achieve the adaptation goal must be as small as possible. Managing resources overshoot is important to avoid also system unstability. This property expresses how well the adaptation performs under given conditions in terms of a balance among the amount of resources used in excess to achieve a shorter settling-time before reaching a stable state [Lu *et al.*, 2000, Appleby *et al.*, 2001, Parekh *et al.*, 2002, Candea *et al.*, 2004, Kumar *et al.*, 2007].

Robustness with respect to context unpredictability. The managed software application services must remain unaltered even if the managed application state differs from the expected state in some measured way. Also, the adaptation process is robust if the adaptation mechanism is able to operate within desired limits even under unforeseen context conditions [Dowling and Cahill, 2004, Meng, 2000].

Termination (of the adaptation process). In software engineering approaches, the MAPE planner typically produces an adaptation plan as a list of discrete controlling actions to modify the managed application. The termination property guarantees that this list is finite and its execution will finish, independently of the final state reached by the managed application. Termination is also referred to as deadlock-free execution, meaning that, for instance, a reconfigurable adaptation process must avoid adaptation rules with deadlocks among them [Ehrig *et al.*, 2010].

Atomicity. In the same context of *termination* (i.e., software engineering approaches), either the system is adapted and the adaptation process finishes successfully or it is not finished and the adaptation process aborts. If an adaptation process fails or aborts, the system is returned to its previous consistent state [Léger *et al.*, 2010].

Consistency. This property aims at ensuring the structural and behavioral integrity of the managed application after performing adaptation processes. For instance, when an adaptation plan is based on dynamic reconfiguration of software architecture, consistency concerns are to guarantee sound interface wiring and bindings between components (e.g., component-based structural conformance) and to ensure that when a component is replaced dynamically by another one, the execution must continue without affecting the function of the affected component. These conditions help protect the application from reaching inconsistent states as a result of, for instance, dynamic reconfigurations based on faulty reconfiguration code or rules [Mukhija and Glinz, 2005, Léger *et al.*, 2010].

Scalability. The capability of an adaptation mechanism to support increasing demands of processing capabilities with sustained performance by using additional computing resources. For instance, scalability is an important property for the adaptation mechanism when it must evaluate an excessive increased number of conditions when analyzing context. As computational efficiency is relevant for guaranteeing performance in the adaptation mechanism, in this case scalability is required to avoid the degradation of any of the operations of the overall system [Appleby *et al.*, 2001, Dowling and Cahill, 2004, Floch *et al.*, 2006].

Security. In a secure adaptation process, not only the managed application but also the data and components shared with the adaptation mechanism are required to be protected from disclosure (confidentiality), modification (integrity), and destruction (availability) [Barbacci *et al.*, 1995].

3.3.2 Quality Attributes and Adaptation Goals on the Managed Application

Besides evaluating adaptation properties, a consistent evaluation of self-adaptive software systems should involve the motivation for building them, that is, their *adaptation goals*. In general, the adaptation can be motivated by the need of continued satisfaction of functional and regulation of non-functional requirements under changing context conditions of execution. Nevertheless, given the goals of this dissertation in the context exposed in the previous section, we focus our analysis on software systems whose adaptation goals are motivated by QoS concerns. Additionally, characteristics of self-adaptive systems such as self-configuring, self-healing or self-optimizing, can be mapped to quality attributes. For instance, following this idea, Salehie and Tahvildari discussed the relationships between autonomic characteristics and quality factors such as the relationship between self-healing and reliability [Salehie and Tahvildari, 2009]. Furthermore, given that adaptation properties sometimes are observed in the managed application, we can use quality attributes to evaluate these properties, while evaluating also managed applications.

In this section we present consolidated definitions of quality attributes that we identified in the analyzed papers, in the context of [Barbacci *et al.*, 1995]. We include corresponding citations of the analyzed contributions that address them. More importantly, based on these quality attributes, in the following section we propose a mapping from adaptation properties to quality factors as a level of indirection for evaluating adaptation properties that are not directly measurable or observable on the adaptation mechanism, but on the managed application.

Performance. Characterizes attributes related to the performance of the services delivered by the managed application, such as timeliness and resource consumption. Timeliness refers to responsiveness, that is, the time required for the application to respond to a specific event or the event processing rate in an interval of time. Resource consumption refers to the added overhead associated to the delivery of a given service, for instance in terms of memory or CPU use. Identified factors that affect performance are latency (the time the application takes to respond to a specific event [Cardellini *et al.*, 2009, Garlan *et al.*, 2004, Mukhija and Glinz, 2005]); throughput (the number of events that can be completed in a given time interval –also called processing rate [Parekh *et al.*, 2002, Dowling and Cahill, 2004, White, 2005, Kumar *et al.*, 2007, Solomon *et al.*, 2010]); and capacity (a measure of the amount of work the system can perform [Parekh *et al.*, 2002, Dowling and Cahill, 2004, Kumar *et al.*, 2007]).

Dependability. Defines the level of reliance that can justifiably be placed on the services the managed application delivers. Adaptation goals related to dependability are availability (readiness for usage [Appleby *et al.*, 2001, Candea *et al.*, 2004, White, 2005, Sicard *et al.*, 2008, Léger *et al.*, 2010, Baresi and Guinea, 2011]); reliability (continuity of service [Floch *et al.*, 2006, Kumar *et al.*, 2007, Sicard *et al.*, 2008, Léger *et al.*, 2010, Baresi and Guinea, 2011, Ehrig *et al.*, 2010]); maintainability (capacity to self-repair and evolve [Appleby *et al.*, 2001, Candea *et al.*, 2004, Floch *et al.*, 2006, Sicard *et al.*, 2008]); safety (from a dependability point of view, non-occurrence of catastrophic consequences from an external perspective (e.g., on the environment) [Baresi and Guinea, 2011]); confidentiality (immune to unauthorized disclosure of information); integrity (non-improper alterations of the system structure, data and behavior [Baresi and Guinea, 2011]).

Security. The selected concerns of the security attribute are confidentiality (protection from disclosure); integrity (protection from unauthorized modification); and availability (protection from destruction [Baresi and Guinea, 2011]).

Safety. The level of reliance that can justifiably be placed on the managed application as no generator of accidents. Safety is concerned with the occurrence of accidents, defined in terms of external consequences. The taxonomy presented in [Barbacci *et al.*, 1995] includes two indicators of critical systems for safety: interaction complexity and coupling strength. In particular, interaction complexity is the extent to which the behavior of one component can affect the behavior of other components. The mentioned taxonomy presents detailed definitions for these two indicators.

3.3.3 Mapping Adaptation Properties to Quality Attributes

We present our mapping from adaptation properties to quality attributes and factors in Table 3.2. As we mentioned before, these quality attributes are referred to both the adaptation mechanism and the managed application, but nonetheless, are aimed at evaluating adaptation properties. For instance, the adaptation property of *consistency* must be verified dynamically (at run-time) in the managed application, using the quality attribute of *dependability*, through its quality factor of *integrity*.

Table 3.2: Mapping adaptation properties to quality attributes and factors

Adaptation Property	Quality Attributes	Quality Factors
Stability	Performance	Latency Throughput Capacity
	Dependability	Safety Integrity
	Security	Integrity
Accuracy	Performance	Latency Throughput
Settling Time	Performance	Latency Throughput
Small Overshoot	Performance	Capacity Resource consumption
Robustness	Dependability	Availability Reliability
	Safety	Interaction Complexity Coupling Strength
Termination	Dependability	Reliability Integrity
Consistency	Dependability	Maintainability Integrity
Scalability	Performance	Latency Throughput Capacity
Security	Security	Confidentiality Integrity Availability

With respect to *stability*, Océano, the dynamic resource allocation system that supports SLAs for peak loads with an order of magnitude of difference, addresses stability based on dependability (i.e., availability and maintainability), and performance (i.e., throughput and capacity—scalability) [Appleby *et al.*, 2001]. In a similar way, the adaptation mechanism proposed by Parekh *et al.* to guarantee desirable performance levels (i.e., throughput and capacity) also addresses stability as an adaptation property [Parekh *et al.*, 2002]. Parekh *et al.* applied an integral control technique to construct a transfer function that models the system and the way the behavior of the managed application is affected by the adaptation mechanism. Baresi and Guinea also addressed stability by proposing a self-recovery system where service oriented architecture (SOA) business processes recover from disruptions of functional and non-functional requirements to avoid catastrophic events (safety) and improper system state alterations (integrity), guaranteeing readiness for service (availability) and correctness of service (reliability) [Baresi and Guinea, 2011].

Concerning *accuracy*, the MOSES framework proposed by Cardellini *et al.* uses adaptation policies in the form of directives to select the best implementation of the composite service according to a given scenario [Cardellini *et al.*, 2009]. MOSES adapts chains of service compositions based on service selection using a multiple service strategy. It has been tested with multiple adaptation goals to observe the behavior of the adaptation strategy in terms of its accuracy (i.e., how close the managed application reaches the adaptation goal). Solomon *et al.* also address accuracy in their self-optimizing mechanism for business processes [Solomon *et al.*, 2010]. They used a simulation model to anticipate performance levels and make decisions about the adaptation process. For this, a tuning algorithm keeps the simulation model accurate. This algorithm compensates for the measurement of actual service time to increase the accuracy of simulations by modeling errors, probabilities and inter-arrival times. Then, it obtains the best estimate for these data such that the square root of the difference between the simulated and measured metrics is minimized.

Regarding *settling time*, Appleby *et al.* measure this property in terms of the time required for deploying a new processing node including the installation and reconfiguration of all applications and data repositories in Océano [Appleby *et al.*, 2001]. Similarly, White evaluated settling time in terms of the average response time required for autonomic EJBs to adapt [White, 2005]. Candea's approach, on another side, applied a recursive strategy that reduces mean time to repair (MTTR) by means of recovering minimal subsets of failed system components. If localized and minimal recovery is not enough, progressively larger subsets are recovered [Candea *et al.*, 2004].

With respect to the last SASO property, the control-based approach by Parekh *et al.* addresses *small overshoot* by avoiding control values (e.g., MAXUSERS) to be set to values that exceed their predicted ranges. They used root-locus analysis based on empirical studies on the transfer function properties to predict the valid values of the maximum number of users to guarantee SLOs. Then, the valid range of values is divided into three regions in order to decide when the control values reach undesirable levels [Parekh *et al.*, 2002].

Dowling and Cahill address *Robustness* in their self-management approach for balancing system load [Dowling and Cahill, 2004]. They aimed at realizing a robust adaptation mechanism by implementing it via decentralized agents that mitigate the negative effects of centralized points of failure.

For verifying *termination*, dynamic and static mechanisms can be used. Ehrig *et al.* proposed a self-healing mechanism for a traffic light system to guarantee continuity of service (reliability) by self-recovering from predicted failures (integrity) [Ehrig *et al.*, 2010]. They addressed termination by statically checking that self-healing rules are deadlock-free, in such a way that the self-repairing mechanism never inter-blocks traffic lights in the same road intersection.

The *scalability* property is analyzed in the K-Components system [Dowling and Cahill, 2004]. To manage it, this approach defines agents with local rules to support self-management and evolving capabilities. Another approach where scalability is addressed as an adaptation property is Madam, the middleware proposed by Floch *et al.* for enabling model-based adaptation in mobile applications [Floch *et al.*, 2006]. Scalability is a concern in Madam for several reasons. First, its reasoning approach might result in a combinatorial explosion if all possible variants are evaluated; second, the performance of the system might be affected when reasoning on a set of a concurrently running applications competing for the same set of resources. Madam proposes an adaptation mechanism where each component (e.g., the adaptation mechanism) can be replaced at run-time to experiment with different analysis approaches for managing scalability.

The *security* adaptation property was not addressed by any of the analyzed self-adaptive systems. Thus, our proposed definition of security as a quality attribute and its corresponding quality factors to evaluate security corresponds to the given in [Barbacci *et al.*, 1995]. As presented in Table 3.1, security of the adaptation mechanism should be evaluated independently of the managed application. This is because ensuring security at the managed application does not guarantee security in the adaptation mechanism.

3.3.4 Towards Adaptation Metrics

Adaptation metrics provide concrete means for evaluating adaptation mechanisms with respect to particular adaptation properties. To identify relevant metrics, we characterize factors that affect the evaluation of quality attributes such as resource usage, throughput, response time, processing rate, mean time to failure, and mean time to repair [Barbacci *et al.*, 1995, Lu *et al.*, 2000]. These factors are essential when considering the metrics to evaluate properties on both the adaptation mechanism and the managed application [Reinecke *et al.*, 2010].

Cardellini *et al.* evaluated performance and reliability in MOSES using the following metrics: expected response time (Ru) (the average time needed to fulfill a request for a composite service); expected execution cost (Cu) (the average price to be paid for a user invocation of the composite service); and expected reliability (Du) (the logarithm of the probability that the composite service completes its task for a user request [Cardellini *et al.*, 2009]).

Appleby *et al.* measured dependability (e.g., availability) and performance factors (e.g., scalability in terms of throughput and capacity) in Océano using the following metrics: active connections per server (the average number of active connections per normalized server across a domain); overall response time (the average time for requests to a given domain to be processed); output bandwidth (the average number of outbound bytes per second per normalized server for a given domain); database response time (average time for requests to a given domain to be processed by the back-end database); throttle rate (T) (a percentage of connections disallowed to pass through Océano on a customer domain); admission rate (the complement of the domain throttle rate $(1-T)$); and active servers (the number of active normalized-servers which service a given customer domain [Appleby *et al.*, 2001]).

Average response time is a common metric used to evaluate performance in several adaptive approaches, such as the framework to develop autonomic EJB applications proposed by White [White, 2005]. Another example is K-Components, which optimizes system performance based on a load balancing function on every adaptation contract that uses a cost function to calculate its internal load cost and the ability of its neighbors to handle the load [Dowling and Cahill, 2004]. This cost function is defined as the sum of the advertised load cost and internal cost of the component (i.e., calculated as the estimated cost to handle a particular load type).

Parekh *et al.*, in their control-based approach to achieve performance service level objectives, used the length of the queue of the in-progress client requests as the metric to control the offered load (the load imposed on the server by client requests) [Parekh *et al.*, 2002]. Baresi and Guinea also proposed a metric to control reliability on the adaptation of BPEL business processes [Baresi and Guinea, 2011], based on the number of times a specific method responds to within two minutes over the total number of invocations. Moreover, they defined a KPI based on this metric, such that reliability must be greater that 95% over the previous two hours of operation.

Kumar *et al.* defined a business value KPI in terms of factors, such as the priority of the user accessing the information, the time of day the information is being accessed, and other aspects that determine how critical the information is to the enterprise [Kumar *et al.*, 2007]. For this, they used a utility function as a combination of factors such as: $utility(e_{gj-k}) = f(\sum d_{ni}, min(b_n i), b_{gj-k})$, where $i|e_{ni} \in M(e_{gj-k})$. The business utility of each edge (e_{gj-k}), which represents data streams between operators that perform data transformations, is a function on the delay d_{ni}, the available bandwidth b_{ni} of the intervening network edges e_{ni}, and the required bandwidth b_{gj-k} of the edge e_{gj-k}.

Candea *et al.* evaluated availability in terms of mean time to recover $(MTTR)$ [Candea *et al.*, 2004]. For this, they defined two metrics: availability $(A = MTTF/(MTTF+MTTR))$ and downtime of unavailability $(U = MTTR/(MTTF+MTTR))$, where $MTTF$ is the mean time for a system or subsystem to fail (i.e., the reciprocal of reliability), $MTTR$ is the mean time to recover, and A is a number between 0 and 1. U can be approximated to $MTTR/MTTF$ when $MTTF$ is much larger than $MTTR$. Similarly, Sicard *et* *al.* defined a metric for availability in terms of $MTTR$ [Sicard *et al.*, 2008].

So far, we have analyzed several metrics that have been used for measuring adaptation mechanisms in different proposals. However, although the identified list of metrics constitute an important base for measuring adaptation properties, it is worth noting that these metrics do not cover the complete list of the adaptation properties.

3.4 The Framework for Classifying Self-Adaptive Software Systems

Our framework to classify and compare self-adaptive software systems consists of the six characterizing dimensions and the ten adaptation properties defined previously, with their corresponding mappings to quality attributes.

Applying the Characterizing Dimensions

We analyzed the selected representative research papers on SAS approaches following the classification options discussed previously for the characterizing dimensions of our evaluation framework. From this analysis we obtain the results presented in Table 3.3. Moreover, based on this analysis we further summarize this characterization in Table 3.4, and discuss its most salient points as follows.

The analyzed approaches range from pure control theory to pure software engineering-based approaches, with several hybrid approaches between them. In the approaches based on control theory, control actions are continuous signals that affect behavioral parameters of the managed application. The structure of the managed application in these approaches is generally non-modifiable, while its behavior is modeled usually with continuous mathematics [Parekh *et al.*, 2002]. In contrast, software engineering-based approaches are characterized by computing sequences of discrete control actions (i.e., *adaptation plans*) that modify the software architecture of the managed application. In these approaches the adaptation is supported by a model of the managed application structure and corresponding reflection capabilities [Appleby *et al.*, 2001, Dowling and Cahill, 2004, Floch *et al.*, 2006, Garlan *et al.*, 2004, Kumar *et al.*, 2007, Léger *et al.*, 2010, Mukhija and Glinz, 2005, Sicard *et al.*, 2008]. In hybrid adaptive systems, control actions are generally discrete operations that affect either the computing infrastructure executing the managed application, or the set of (operating system level) processes comprising it. In some of these cases, the structure of the managed application is even non-modifiable [Baresi and Guinea, 2011, Candea *et al.*, 2004, Cardellini *et al.*, 2009, Ehrig *et al.*, 2010, White, 2005]. In some others, such as the proposed in [Solomon *et al.*, 2010], parameters for dynamic predictive models are dynamically tuned, based on simulations, but the analysis to decide whether to adapt it is based on a continuous model of its behavior. Although discrete control actions to affect the architecture of the managed application are implied, the paper does not explain how these are produced. This kind of approaches, which mix strategies in varied ways, further expand the classification spectrum for hybrid approaches. Concerning *when* these controlling actions (or adaptation plans) are produced, in most of the cases they are produced statically (e.g, coded by hand at design-time).

Concerning the reference control inputs (i.e., the target objective states to be reached by the adaptation mechanism), most approaches use some abstracted form of logical conditions, such as contracts, or expected values on utility functions. These specifications correspondingly determine the measured outputs on the managed application. All approaches that explicitly addressed monitoring on these outputs, specify monitor probes on logical attributes of the managed application elements (internal context), while two of them take the external context into account [Kumar *et al.*, 2007, Mukhija and Glinz, 2005]. Regarding the adaptation mechanism

Table 3.3: Characterizing dimensions applied to selected SAS approaches

Approach	Adaptation Goal	Reference Inputs	Measured Outputs	Control Actions	System Structure	Adaptation Properties
Appleby *et al.* Océano, 2001	Self-manag. Self-optim.	Contracts: SLAs	SLOs/Logical properties of computational elements	Discrete operations affecting the comput. infrastructure	Adaptive ctrl./Modif. struct.reflect.	Stab., settl. time, small overshoot, scalab./Scalab., Dependability: availab., maintain.
Baresi and Guinea, 2010	Self-recov.	Contracts: SLAs, funct. req.	SLOs/Logic. prop. of comput. elem.	Discrete oper. managed applic. process	Adaptive ctrl./Non-modif.struct.	Controller: None/ Behav., Dependab: safety, integrity, availab., reliab.
Candea *et al.* Microreboots, 2004	Self-recov.	Contracts: SLOs-QoS	Malfunct. cond. /Logic.prop. of comput. elem.	Discrete oper. managed applic. process	Adaptive ctrl./Modif. struct.reflect.	Overshoot, settl. time/Dependab: availability
Cardellini *et al.* MOSES, 2009	QoS preser.	Contracts: QoS	SLOs/Logic. properties of comput. elem.	Discrete oper. managed applic. process	Reconfig. ctrl./Modif. struct.reflect.	Accuracy/ Perform.: latency, Depend.:reliab., cost
Dowling and Cahill. K-Comp., 2004	Self-manag.	Contracts: SLOs-QoS	SLOs/Logic. properties of comput. elem.	Discrete oper. managed applic. soft. architecture	Reconfig. ctrl./Modif. struct.reflect.	Robust., scalab./ Performance: througp., capac.
Ehrig *et al.*, 2010	Self-healing	Goal actions	Malfunct. condit./Logic. properties of comput. elem.	Discrete oper. managed applic. process	Adaptive ctrl./Non-modifiable structure	Termination/ Dependab:reliab.
Floch *et al.* MADAM, 2006	QoS preser. Self-config.	Contracts: SLAs-SLOs-QoS	SLOs/Logic. properties of comput. elem.	Discrete oper. managed applic. soft. architecture	Adaptive ctrl./Modif. structure reflection	Scalab/Dependab: reliab., maintain.
Garlan *et al.* Rainbow, 2004	Self-repair.	Contracts: SLAs-SLOs-QoS	SLOs/Logic. properties of comput. elem.	Discrete oper. managed appl. soft. arch.	Adaptive ctrl./Modif. struct.reflect.	None for the controller/Perform: latency
Kumar *et al.* MWare, 2007	Self-manag. Self-config. Self-optim.	Contracts: QoS; policy actions	SLOs/Logic. properties of comput. elem. ext. context	Discrete oper. managed applic. soft. architecture	Adaptive ctrl./Modif. structure reflection	Settl. time, overshoot/ Performance: through., capac.
Léger *et al.*, 2010	QoS preser. Self-config.	Constraints: comput. states	Malfunct. condit./Logic. properties of comput. elem.	Discrete oper. managed applic. soft. architecture	Reconfig. ctrl./Modif. struct.reflect.	Consist.: atomic., isol., durab./ Dependab.:availab., reliability
Mukhija and Glinz CASA, 2005	QoS preser. Self-config.	Contracts: QoS	SLOs/Logic. properties of comput. elem. ext. context	Discrete oper. managed applic. soft. architecture	Reconfig. ctrl./Modif. structure reflection	Consistency/ Performance
Parekh *et al.*, 2002	QoS preser.	Single reference value	SLOs/Logic. properties of comput. elem.	Continuous signals behavioral properties	Feedback ctrl./ Non-modif.struct. math.model	Stab., overshoot/ Performance: throughput, capacity
Sicard *et al.*, 2008	Self-manag. self-healing	Constraints: comput. states	Not explicit monitoring	Discrete oper. managed applic. soft. architecture	Feedback ctrl./Modif. structure reflection	None for the controller/ Dependab.: reliab., avail.
Solomon *et al.*, 2010	Self-optim.	Business KPIs	KPIs/Logic. properties of comput. elem.	Not addressed (simulation model)	Adaptive ctrl./ Modif. structure reflection	Accuracy/ Performance
White. Autonomic JBeans, 2005	Self-manag. Self-healing Self-protect.	Contracts: SLOs-QoS	SLOs/Logic. properties of comput. elem.	Discrete oper. managed applic. process	Adaptive ctrl./ Non-modif. structure	Settl. time/ Perform.:through., Depend.: availab.

structure, all approaches, except [Parekh *et al.*, 2002] and [Sicard *et al.*, 2008] that implement a simple feedback loop, implement either adaptive or reconfigurable control mechanisms.

3.5 Chapter Summary

Self-adaptation has increasingly become a fundamental concern in the engineering of software systems. On one side, it is a factor to reduce the high costs of software maintenance and evolution; on the other side, it used to regulate the satisfaction of functional and extra-functional requirements under changing conditions of system execution. Even though adaptation mechanisms have been widely investigated in the engineering of dynamic software systems, their application to actual problems in industry is still limited due to a lack of methods for validation and verification of the complex and nonlinear software systems.

In this chapter we analyzed a representative subset of research works on self-adaptive software (SAS) proposals. From this analysis, we identified a divergent and difficult-to-compare set of methods to assess this kind of systems. Some validation mechanisms discussed in these approaches are limited to the performance evaluation of the managed application, even when the adaptation goal is not related to this quality attribute. Nonetheless, we were able to consolidate the definitions for a set of identified properties inherent to SAS systems, and proposed a mapping from them to quality attributes. We also discussed how these properties help to leverage the benefits of self-adaptation in a variety of ways. For instance, one important application of them is in the realization of characteristics such as the reliability and trustworthiness of this kind of autonomous systems.

Finally, in our opinion, the framework for classifying and comparing SAS systems that we defined in this chapter constitutes a starting point for discussing on our proposed adaptation properties. These properties, as inherent to SAS systems, provide a common basis to assess adaptation mechanisms specially when this assessment has comparative, improvement, or even mix-and-match combination purposes.

In the next chapter, we present the formal definitions that constitute the foundation of our model for dynamic reconfiguration to preserve QoS contracts. We also use the videoconference system of our example scenario introduced in the previous chapter to illustrate these definitions.

Table 3.4: Selected SAS systems characterization summary

Characteristic	Count	[List of Approaches]
		Spectrum Classification
Control Engineering	1	[Parekh *et al.*, 2002]
Hybrid	5	[Baresi and Guinea, 2011, Candea *et al.*, 2004, Cardellini *et al.*, 2009, Ehrig *et al.*, 2010, White, 2005]
Hybrid-Software	1	[Solomon *et al.*, 2010]
Software Engineering	9	[Appleby *et al.*, 2001, Dowling and Cahill, 2004, Floch *et al.*, 2006, Garlan *et al.*, 2004, Kumar *et al.*, 2007, Léger *et al.*, 2010, Mukhija and Glinz, 2005, Sicard *et al.*, 2008]
		Monitoring Mechanisms
Monitor internal context	13	
Monitor external context	2	[Kumar *et al.*, 2007, Mukhija and Glinz, 2005]
Not specified	1	[Sicard *et al.*, 2008]
		Adaptation Mechanism Structure
Feedback control	2	[Parekh *et al.*, 2002, Sicard *et al.*, 2008]
Adaptive control	9	[Appleby *et al.*, 2001, Baresi and Guinea, 2011, Candea *et al.*, 2004, Ehrig *et al.*, 2010, Floch *et al.*, 2006, Garlan *et al.*, 2004, Kumar *et al.*, 2007, Solomon *et al.*, 2010, White, 2005]
Reconfigurable control	5	[Cardellini *et al.*, 2009, Dowling and Cahill, 2004, Léger *et al.*, 2010, Mukhija and Glinz, 2005]
		Managed Application Structure
Non-modifiable	4	[Baresi and Guinea, 2011, Ehrig *et al.*, 2010, Parekh *et al.*, 2002, White, 2005]
Modifiable with reflection	12	[Appleby *et al.*, 2001, Candea *et al.*, 2004, Cardellini *et al.*, 2009, Dowling and Cahill, 2004, Floch *et al.*, 2006, Garlan *et al.*, 2004, Kumar *et al.*, 2007, Léger *et al.*, 2010, Mukhija and Glinz, 2005, Sicard *et al.*, 2008, Solomon *et al.*, 2010]
		Adaptation Properties
Settling time	4	[Appleby *et al.*, 2001, Candea *et al.*, 2004, Kumar *et al.*, 2007, White, 2005]
Small overshoot	4	[Appleby *et al.*, 2001, Candea *et al.*, 2004, Kumar *et al.*, 2007, Parekh *et al.*, 2002]
Scalability	3	[Appleby *et al.*, 2001, Dowling and Cahill, 2004, Floch *et al.*, 2006]
Stability	2	[Appleby *et al.*, 2001, Parekh *et al.*, 2002]
Accuracy	2	[Cardellini *et al.*, 2009, Solomon *et al.*, 2010]
Termination	1	[Ehrig *et al.*, 2010]
Atomicity	1	[Léger *et al.*, 2010]
Consistency	3	[Léger *et al.*, 2010, Mukhija and Glinz, 2005]
Robustness	1	[Dowling and Cahill, 2004]
Security	0	
		Quality Attributes
Performance	10	[Appleby *et al.*, 2001, Cardellini *et al.*, 2009, Dowling and Cahill, 2004, Floch *et al.*, 2006, Garlan *et al.*, 2004, Kumar *et al.*, 2007, Mukhija and Glinz, 2005, Parekh *et al.*, 2002, Solomon *et al.*, 2010, White, 2005]
Dependability	7	[Appleby *et al.*, 2001, Baresi and Guinea, 2011, Candea *et al.*, 2004, Cardellini *et al.*, 2009, Ehrig *et al.*, 2010, Floch *et al.*, 2006, White, 2005]

Chapter 4

A Formal Model for QoS Contracts-Preserving Reliable Reconfiguration

Contents

In this chapter we present our formal model for reliable preservation of QoS contracts in component-based software applications. To accomplish this, we model the dynamic reconfiguration of these software applications as an action associated to the notification of events that (potentially) violate their contracted QoS levels of operation on changing execution conditions. We conceive this formalization as a sound foundation for software applications to be autonomously responsible for their QoS contracts, as envisioned in Component-Based Software Engineering (CBSE) [Szyperski, 1998, Heineman and Councill, 2001].

To develop our model we consider the principles and concerns analyzed and explored by the Software Engineering for Adaptive and Self-Managing Systems research community—SEAMS—in [Cheng *et al.*, 2009a] and [de Lemos *et al.*, 2012]. These are mainly related to the separation of concerns between managed applications and adaptation mechanisms, the explicitness of feedback-loop elements in the self-adaptive system architecture, the enforcement of adaptation properties, and the management of context unpredictability.

In our vision, QoS contracts establish minimum expected *levels of operation* (QoS levels) that must be satisfied and preserved by a managed software application. These QoS levels are specified for each of the different context conditions to be faced by the managed application in its execution. Thus, the continuous satisfaction of a QoS contract (i.e., its preservation) implies to satisfy each of these QoS levels that the user expects, under each of the corresponding varying conditions of execution. At runtime, once these conditions actually occur in the execution context of the managed application, the respective QoS levels must be monitored, and their fulfillment, enforced.

Hence, these QoS levels, expressed as first-order logic predicates, can be seen as *context-dependent* "system invariants", following the concept of program invariants given by [Hoare, 1969] (i.e., predicates that are true in the context of a specific sequence of instructions). In other words, these expected QoS levels defined by users in QoS contracts specify particular properties (context-dependent system invariants) required to be continuously satisfied by managed applications as context conditions change. Examples of these properties are context-dependent conditions on throughput, response time, confidentiality, and readiness of the managed application services.

However, guaranteeing the continuous and trustworthy satisfaction of these user-defined properties through self-reconfiguration is a challenging problem. On one hand, producing reconfiguration plans to satisfy these properties, traditionally coded by hand at design-time for specific context conditions and application states, is a difficult and error-prone task. The difficulty is not only in tracking the different elements to deploy/undeploy and (re)wire/unwire, but in coding these plans to match the different evolved managed application states, specially under varying execution conditions. On the other hand, in contrast to the particular QoS properties required by the user for a managed application, the properties to be certified for the reconfiguration mechanism should be the inherent properties that characterize the own nature of self-adaptive software systems. These are the *adaptation properties* that we defined in the previous chapter, which should be intrinsic in the reconfiguration mechanism.

In light of these problems, in this chapter we address two fundamental challenges:

(i) To devise context-aware reconfiguration mechanisms that continuously operate to fulfill user-defined QoS levels under varying conditions of execution, by applying user-defined reconfiguration rules. The reconfiguration mechanism must manage a number of components with their interrelationships, considering all differences between the actual vs. the next configuration states. It is worth noting that part of this challenge is that both, the required QoS properties and also the reconfiguration rules must be defined by the user, in QoS contracts. Nonetheless, as a consequence, the effectiveness of achieving the respective QoS levels is determined by the effectiveness and adequacy of the user-defined reconfiguration rules. This requires from the reconfiguration mechanism to address questions such as what would happen if the user (e.g., a software evolution architect) mistakenly provides irrelevant rules for guaranteeing a given QoS level? would the system loop endlessly applying these rules, never fulfilling the target QoS level besides wasting computing resources? would the user be notified about this anomalous situation? Moreover, QoS contracts must be preserved in continuous reconfiguration loops at runtime, starting and ending in actual running software applications, and also considering dynamic changes in contracts.

(ii) To guarantee the reliability as an inherent characteristic of the reconfiguration mechanism, independently of the managed application's QoS properties specified in contracts, and transparently to the end-user. We define this reliability in terms of the adaptation properties of atomicity, termination, short settling-time, structural consistency, and robustness with respect to context unpredictability, that we characterized previously. It is worth noting that even though we defer the proofs showing that our reconfiguration mechanism guarantees these properties to Chapter 6, we address this challenge specifically by using formal models that adequately support them.

Therefore, to address these challenges, we build our formal model as follows. On one hand, our choice of Typed Attributed Graph (called *e-graphs*) Transformation System theory (*TAGTS*) [Ehrig *et al.*, 2009] to model software structure configuration and its dynamic re-configuration allows us to exploit *design-patterns* at runtime. We use these patterns in reconfiguration loops to fulfill the expected QoS levels at runtime—just as they are used for the same purpose at design time—while the software reconfiguration benefits from graph transformation properties. Design patterns embody sets of components whose joint operation, as a whole, determine the QoS levels of the services in which they participate. On the other hand, we extend Finite State Machines (*FSM*) [Hopcroft *et al.*, 2006] to govern these reconfiguration loops, not only to manage contract fulfillment and *unfulfillment* states, but also to maintain the association of these states to the actual running-software state, under the current context situation.

This chapter is organized as follows. In Section 4.1 we give an overview of the strategy that we follow in our formal model to continuously fulfill QoS levels under varying execution conditions. In Sections 4.2 and 4.3 we respectively present our e-graph based model for reliable reconfiguration and FSM based robust management of QoS contracts and managed application states. Finally, it is worth clarifying that while this chapter presents our formal model, in the next chapters we explain how do we realize it, and analyze its corresponding adaptation properties.

Correspondences in this Chapter: *Addressed Challenge(s):* C2; C3; C4 –Reconfiguration plans to preserve QoS contracts must be generated automatically and dynamically from parameterized reconfiguration rules defined by users; The reconfiguration mechanism must be clearly separated from the target software application, as well as their corresponding properties; Uncertainty must be managed robustly with respect to the unpredictability of context events faced by the target application, as well as the parameterized reconfiguration rules in the reconfiguration mechanism. *Goal(s):* G2, G3, G4 –Develop a formal model for the dynamic and reliable reconfiguration mechanism to preserve QoS contracts in component-based software; Maintain a clear separation of concerns between the reconfiguration mechanism and the target software application, as well as between their corresponding properties; Guarantee robustness in the reconfiguration mechanism with respect to possible and foreseeable situations associated to the management of the unpredictable nature of context. *General contribution(s):* GC.FM1, GC.FM2 –E-Graph (TAGTS) based model for reliable preservation of QoS contracts through self-reconfiguration; FSM based model for autonomous and robust management of QoS contracts states. *Specific contribution(s):* FM1.1, FM1.2, FM1.3, FM2.1 –Unified e-graph and machine processable specifications for (i) component-based structures (CBS), (ii) QoS contracts to be satisfied, and (iii) reconfiguration rules; Automatic derivation of reliable reconfiguration plans from e-graph transformations; Independent and reusable reconfiguration mechanism modeled using e-graphs as an abstract and neutral representation for the component-based target application; Characterization of generic QoS contracts states (fulfillment, violation and exception) and transitions, automatically managed at runtime by the FSM based model.

4.1 Overview of the Formal Model in the Solution Strategy

As we analyzed in Section 2.2, service-component platforms execute software applications that provide software services through interfaces implemented by software components subject to QoS contracts. Typically, QoS contracts specify context-dependent QoS levels on a QoS property, such as the confidentiality, illustrated in Table 4.1(a)[23]. This contract specifies three QoS levels for the confidentiality attribute to be guaranteed to the user under given context conditions. The first row in the table specifies that whenever connected from an intranet-serviced area, the user can expect to have a clear-channel communication for the corporate videoconference services; meanwhile the second, that whenever she moves to an extranet-serviced area, she can expect a secure-channel, dynamically configured in a transparent way for her. It is worth noting that in this simple example the contract could be renegotiated to have finer-grained confidentiality requirements. For instance, instead of simply specifying a confidential channel for extranet connections, we could require 64bit ciphering for extranet UMTS connections, while 1024bit for extranet WiFi connections.

A simplified service-component videoconference application subject to this contract is illustrated in Fig. 4.1 running in the context of a corporate building, that is, being able to register in and attend a videoconference using a clear channel connection.

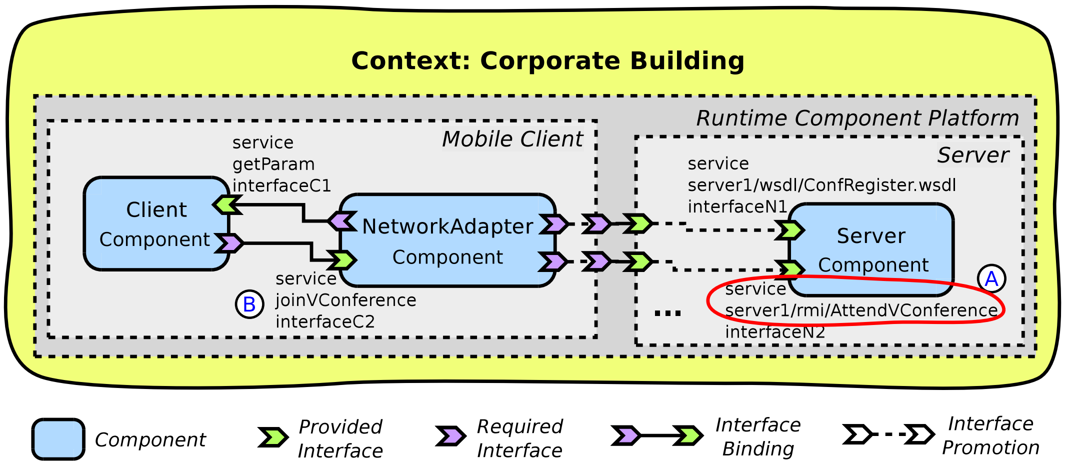


Figure 4.1: Simplified service-component videoconference application in the context of the corporate building. Components provide services (through interfaces) that can be required by others. Connections from required to provided services (interfaces) specify the corresponding services invocations. In particular, (A) highlights the RMI service *AttendVConference* provided by the *Server* component for a user to attend a videoconference. This service is invoked by the *Client* component through (B) the *joinVConference* service of the *NetworkAdapter* component.

In this initial situation the software application is of course fulfilling the agreed QoS contract, as specified by the QoS level *Clear Channel* under the context condition *C1:Intranet Connection* in the first row of the table representing the contract.

However, when the user goes to the parking lot to wait for transportation to the airport, the videoconference application client confronts a new context condition. This new condition, which is identified as $C2$ in the table representing the contract, is signaled by the change of the network access point (entering an extranet-serviced area) thus requiring the use of a *Confident Channel*, as specified in the contract. In this situation, the software application is expected to perform an autonomous and transparent reconfiguration to fulfill the new contractual condition

[23]This QoS contract was introduced in our example scenario. However, we reproduce it here for readability.

Table 4.1: Context conditions and corresponding QoS levels for:

(a) Confidentiality (based on corporate network access)

Context Condition	Service Level Objective
C1: Intranet Connection	Clear Channel
C2: Extranet Connection	Confident Channel
C3: No Netw. Connection	Local Cache

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

Context Condition	Service Level Objective
C4: $Bandwidth \leq 12$	Call on Hold
C5: $12 < Bandwidth \leq 128$	Voice Call
C6: $128 < Bandwidth$	VoiceVideo Call

under $C2$, *before compromising* the current QoS level of $C1$, as illustrated in Fig. 4.2. In effect, in the lower part of this figure we can observe the deployment of *EnDeCipher* components on the corresponding client and server sides. This component symmetrically enciphers and deciphers the transmitted data, guaranteeing a confident communication channel.

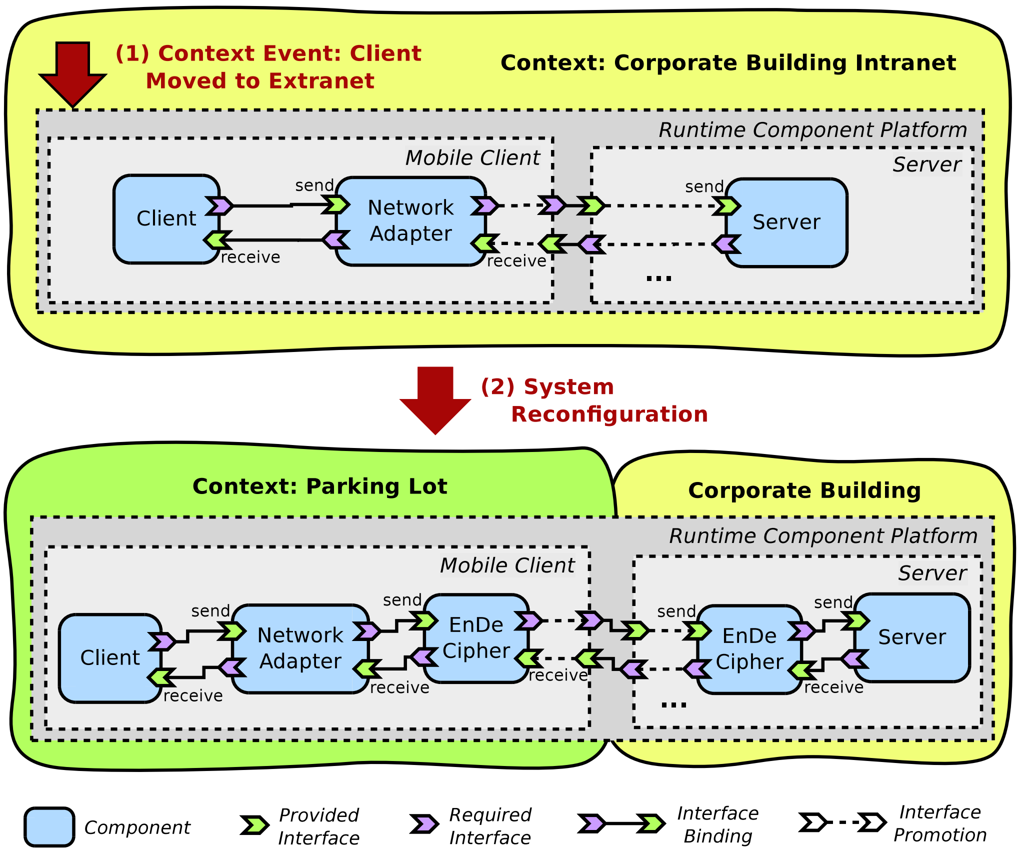


Figure 4.2: Reconfiguration of the service-component application client. When the service-component application client is (1) notified by a change of context condition entering an extranet-serviced area, the user expectation is (2) a self-reconfiguration to preserve the QoS contract.

Nonetheless, as we analyzed in previous chapters, a still open and major challenge for component platforms is the autonomous and reliable preservation of QoS contracts at runtime under varying execution conditions, as illustrated in the example. Our strategy to solve this problem is to use (i) e-graphs to model the architecture of service-component software applications, reconfiguration rules, and QoS contracts; and (ii) finite state machines to control the states of contract fulfillment and unfulfillment on the software application. We perform a state transition whenever a context event arrives notifying of changes in context conditions

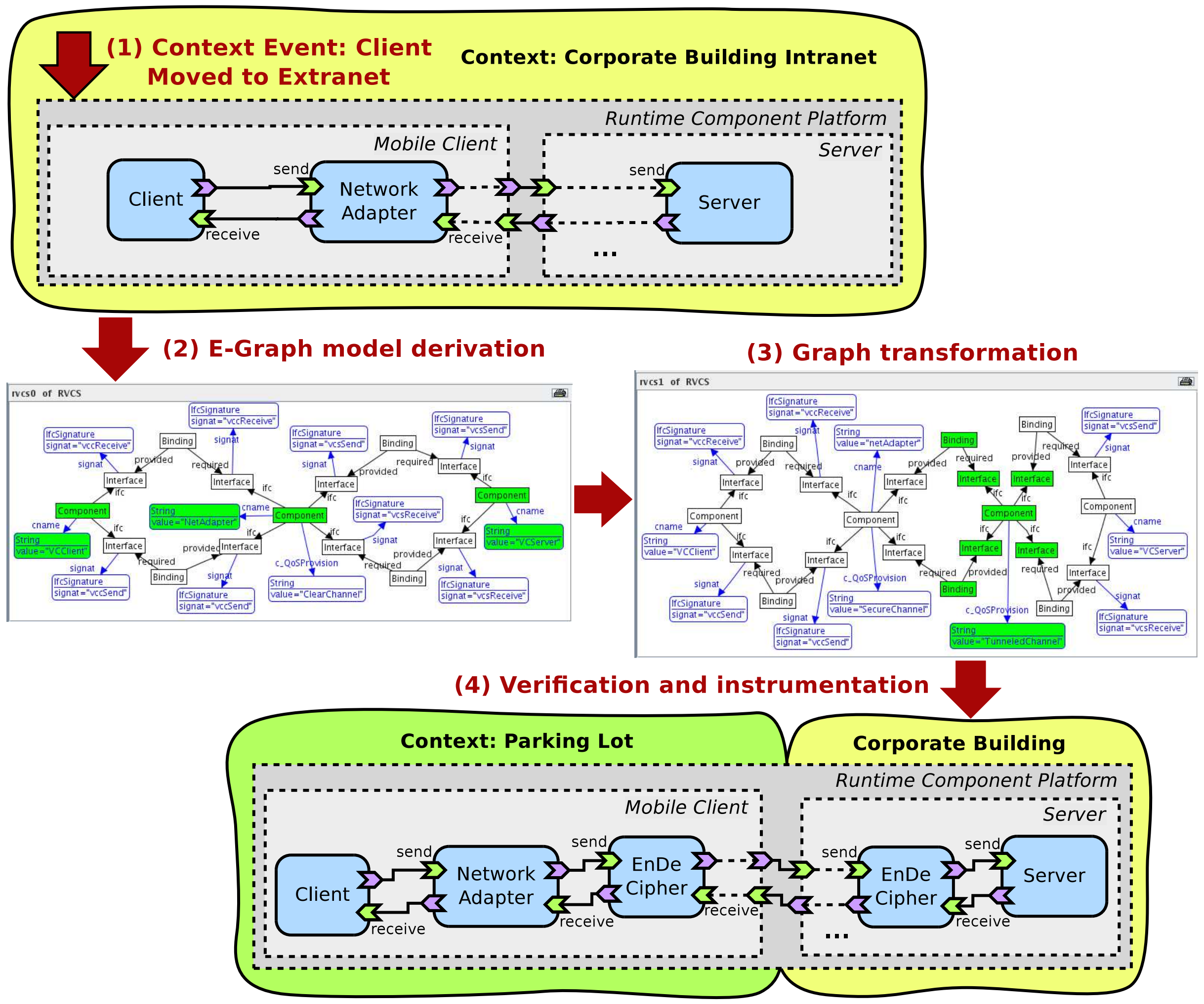


Figure 4.3: Service-component application reconfiguration as graph transformation.

that may violate the current QoS level. Given that states correspond to structural configurations of the software application represented in e-graphs, a state transition is a graph transformation operation based on graph rewriting rules. To specify each of these rules, we encode design patterns in its left and right hand sides, benefiting from the well known relationships between these patterns and software QoS properties [Shaw and Garlan, 1996, Bass *et al.*, 2003, Kruchten *et al.*, 2006, Buschmann *et al.*, 2007, Clements and Shaw, 2009, Ramachandran, 2002, Zeng *et al.*, 2004b, Kircher and Jain, 2004, Krakowiak, 2009, Dougherty *et al.*, 2009]. That is, left-hand sides are used to identify software substructures in managed applications by graph-based pattern matching. Corresponding right-hand sides specify the target structures into which the identified application substructure must be reconfigured, by graph transformation. In the illustrated example, we use a rule that reconfigures the application structure when moving from an intranet to an extranet-serviced area. Even though we defer the illustration of this reconfiguration rule to the next section, we anticipate that this rule uses a clear-channel structure pattern as its left-hand side, and a confident-channel structure pattern as its right-hand side. Thus, the effect of applying this rule in a software application is to transform its clear communication channels that match the rule left-hand side, into confident communication channels.

What we consider most important in our approach, in contrast to other formal based approaches, is that we carefully ground the objective of our formal modeling on actual running software systems. That is, based on the service-component architecture (SCA) specification and the introspection capabilities of component platforms, we provide a function that synthesizes our corresponding e-graph representation. In fact, we actually invoke this function from the component platform in order to apply our formal model on the running state of service-component applications (marked (2) in Fig. 4.3). Thus, we dynamically reconfigure the software application structure by first performing a graph transformation on its e-graph representation (marked (3) in the figure). Only if the resulting graph structure successfully passes the verification checks, such as the structural components/connectors conformance, we instrument the reconfiguration modifications in the actual running software application (marked (4) in the figure). For this instrumentation we require at least the provision of reconfiguration primitives by the component platform, such as those provided by the FraSCAti middleware [Seinturier *et al.*, 2009], which we use in this work. The adaptation processes, which begin and end in the actual service-component software application executed by the component platform, are governed by the finite state machine representing the contracted QoS levels.

Finally, formal models entail benefits that non-formal models are unable to confer. In our case, we base our model on e-graphs and finite state machines to formally guarantee the reliability of the adaptation process in terms of adaptation properties, as defined in the previous section. A second important benefit is that we are able to define a clear semantics for our approach, from which algorithms can be derived, preserving the formal model properties.

4.2 E-Graph Modeling of QoS Contracts-Driven Reconfiguration

Our Assumption A1 enunciated in Section 1.3 recalled the determining relationship between software design patterns and specific software QoS levels. We use this relationship as the cornerstone of our strategy to preserve QoS contracts through dynamic reconfiguration as follows:

i. QoS contracts comprise different QoS levels for corresponding context conditions on specific software quality attributes;

ii. specific design patterns, known to address the quality attributes implied by QoS contracts, can be encoded in left and right hand sides of reconfiguration rules, depending on the desired effects;

iii. left-hand sides are used to identify actual managed application substructures associated to actual QoS levels;

iv. corresponding right-hand sides specify the target substructures into which the identified managed application substructure must be reconfigured, whenever a context change occurs;

v. reconfiguration rules specify the mappings between left and right-hand sides, and more specifically, which components to introduce, which to remove, and how to (re)connect them;

vi. for each QoS level to be achieved, the user (e.g., a system evolution architect who had previously negotiated the contract with the client) specifies the respective reconfiguration rules.

In essence, these statements correspond to a rule-based graph transformation problem, if we represent the structure of managed applications as graphs, and reconfiguration rules as graph-rewriting rules. To identify design patterns (encoded as graphs in left-hand sides) in actual managed applications, we use graph-based pattern matching. Then, we transform the managed application structures into target design-pattern structures using a reconfiguration system built on a graph transformation system. This process is driven by a QoS contract definition (and corresponding semantics in extended state machines) that we tailored for our graph-based model. From this process and considering the actual evolved structure of the managed application and the current context conditions, we dynamically synthesize reconfiguration plans from the application of the reconfiguration rules.

Moreover, for a CBD[24] system to be responsible for its QoS contracts automatically, it requires (i) to have a structural representation of itself at the component level (i.e., to be reflective) [Cheng *et al.*, 2009a]; (ii) to have a representation of the contracted QoS levels under the different context conditions; (iii) to be self-monitoring, that is, to identify and notify events on the QoS level violations; and (iv) to apply the dynamic reconfiguration in response to events notifying imminent violation of QoS levels, as specified in the QoS contracts.

Based on the previous considerations, we build our e-graph based model for reconfiguring component-based applications on the TAGTS theory developed by [Taentzer, 2004] and [Ehrig *et al.*, 2009], as follows. In Section 4.2.1 we recall the base definitions of e-graphs given in [Ehrig *et al.*, 2009]. Based on the e-graph as a unified formalism, we give our definitions for component-based software reflection structure and QoS contracts respectively in Sections 4.2.2 and 4.2.3. Finally, in Section 4.2.4 we present our component-based structure reconfiguration system. Along these sections, we exemplify how these constructs support software reflection and QoS contracts preservation through self-reconfiguration.

4.2.1 Extended Graphs: Base Definitions

Definition 4.1 (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);*
- $source_1 : E_1 \to V_1$; $source_2 : E_2 \to V_1$; $source_3 : E_3 \to E_1$ *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.*

An e-graph (typed attributed 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, as depicted in Fig. 4.4.

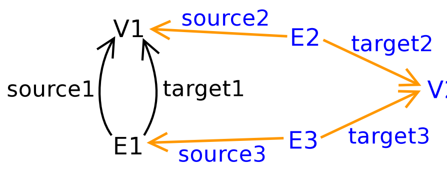


Figure 4.4: E-Graph definition illustration.

Definition 4.2 (E-Graph morphism). *An e-graph morphism f between e-graphs G and H, $f : G \to H$, is a tuple* $(f_{V_1}, f_{V_2}, f_{E_1}, f_{E_2}, f_{E_3})$ *where* $f_{V_i} : G_{V_i} \to H_{V_i}$ *and* $f_{E_j} : G_{E_j} \to H_{E_j}$ *for* $i = 1, 2,\ \ j = 1, 2, 3$, *such that f commutes with all source and target functions*[25]*).*

[24]In the following, we use CBD, CBSE and SCA interchangeably.

[25]E-Graphs combined with E-Graph morphisms form the category $EGraphs$. See [Ehrig *et al.*, 2009] for more details on this.

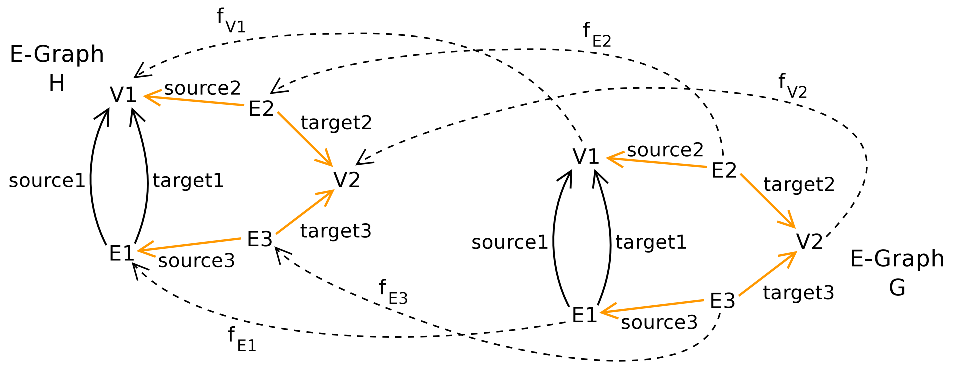


Figure 4.5: E-Graph morphism example between e-graphs G and H, $f : G \rightarrow H$. E-graph morphisms are used as typing relationships between e-graphs.

4.2.2 System Reflection Structure

For a software system to self-reconfigure at runtime, it is required to be reflective. That is, it must be able to identify and keep track of its individual elements that are to be involved in reconfiguration operations [Cheng *et al.*, 2009a]. We base our representation of system reflection structure on the component-based structure definition.

Definition 4.3 (Component-Based Structure –CBS). *The component-based structure, CBS, is the tuple $(G, DSig)$, where:*

- *$DSig$ is a data signature over the disjoint union $Integer + String + IfcSignature + IfcRole + IntPair$ where $IfcRole = \{Provided, Required\}$ and $IntPair = Integer \times Integer$, with the usual CBD 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:*
  - $V_1 = \{Component, Interface, Binding\}$;
  - $V_2 = Integer + String + IfcSignature + IfcRole + IntPair$;
  - $E_1 = \{ifcp, ifcr, provided, required\}$;
  - $E_2 = \{cname, iname, role, signat, c\_QoSProvision, i\_QoSProvision, ct\_QoSProvision, cmultiplicity, imultiplicity, bmultiplicity\}$;
  - $E_3 = \{ifcpmult, ifcrmult, pmult, rmult\}$;
  - $source_1 = \{(ifcp, Component), (ifcr, Component), (provided, Binding), (required, Binding)\}$;
  - $target_1 = \{(ifcp, Interface), (ifcr, Interface), (provided, Interface), (required, Interface)\}$;
  - $source_2 = \{(cname, Component), (iname, Interface), (role, Interface), (signat, Interface), (c\_QoSProvision, Component), (i\_QoSProvision, Interface), (ct\_QoSProvision, Binding), (cmultiplicity, Component), (imultiplicity, Interface), (bmultiplicity, Binding)\}$;
  - $target_2 = \{(cname, String), (iname, String), (role, IfcRole), (signat, IfcSignature), (c\_QoSProvision, String), (i\_QoSProvision, String), (ct\_QoSProvision, String), (cmultiplicity, Integer), (imultiplicity, Integer), (bmultiplicity, Integer)\}$;
  - $source_3 = \{(ifcpmult, ifc), (ifcrmult, ifc), (pmult, provided), (rmult, required)\}$; *and*
  - $target_3 = \{(ifcpmult, IntPair), (ifcrmult, IntPair), (pmult, IntPair), (rmult, IntPair)\}$.

The Component-Based Structure is an e-graph where graph nodes represent the usual CBD's component, interface, interface type and binding elements. Composites (i.e., aggregates of components) are abstracted as components, as we address structural reconfiguration at the system

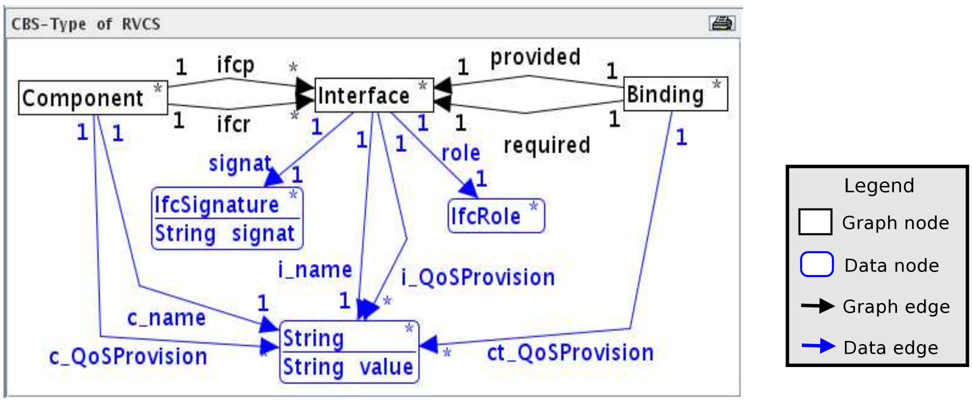


Figure 4.6: E-Graph Component-Based Structure (CBS) illustration. The *Integer* type, *mult* and *multiplicity* attributes, for multiplicity constraints, are represented implicitly in the usual (short) notation.

level. Graph edges correspond to the relationships among these elements. Data nodes represent types of attributes meanwhile the data edges, the typing relationships. **_QoSProvision* are special attributes to annotate components, interfaces and bindings to express their provision of particular QoS capabilities, such as *secure* network connections or *scalable* processing possibilities.

Definition 4.4 (Component-Based Software Application Reflection –CBSAR). *Given S the computational state of a running component-based software application, its corresponding Component-Based Software Application Reflection (CBSAR) state, $\Re_S$, is defined as $\Re_S = (G, f_S, t)$, 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, being S the representation of the state of the managed application components, interfaces and bindings, as maintained in SCA runtime component platforms, $\Re_S$ encapsulates its corresponding e-graph representation. The purpose of f_S is to map the component application structure to the e-graph domain, in which the reconfiguration is to be operated. This definition is crucial for the practical applicability of our proposal, given that it allows the straightforward definition of f_S, based on the component platform introspection capabilities. Once the abstract representation is reconfigured, we use f_S^{-1} to perform the reconfiguration in the actual runtime component-based application. The feasibility of f_S results from Def. 4.3 (CBS) and the e-graph morphism t. $\Re_S.Component$ denotes the set of components in $\Re_S$, that is, $\Re_S.Component = \{c \,|\, c \in G_{V_1} \wedge t_{V_1}(c) = Component\}$ (analogously for the other $CBSAR$ elements).

Example 4.1 (Videoconference Component-Based Structure and Reflection). *Figure 4.7 illustrates the runtime component-based application structure of our RVCS example scenario introduced in Section 2.5 (page 40) when configured to be connected from the intranet (i.e., with a clear-channel connection). The components in this figure are represented in Fig. 4.8 as exactly the videoconference client ($VCClient$) with its network adapter component ($NetAdapter$) and the server ($VCServer$). The other elements represent their interfaces ($vccReceive$, $vccSend$ and so on), and respective bindings. The NetAdapter component, providing a network connection, is responsible for maintaining a ClearChannel connection, as expressed by its c_QoSProvision attribute.*

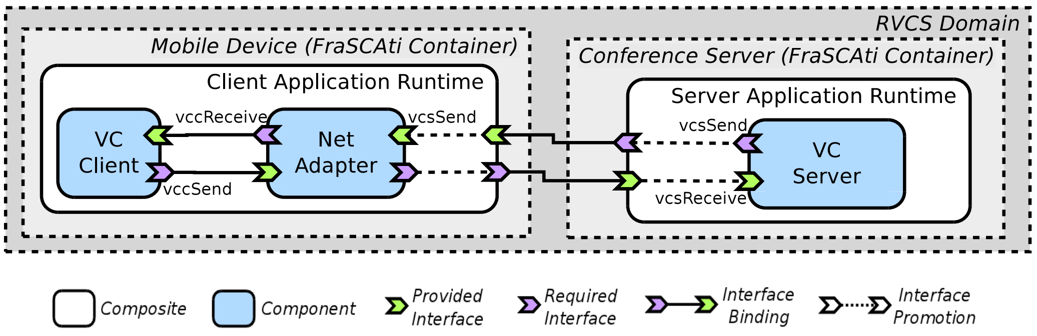


Figure 4.7: Runtime application structure for the RVCS example with a clear channel connection.

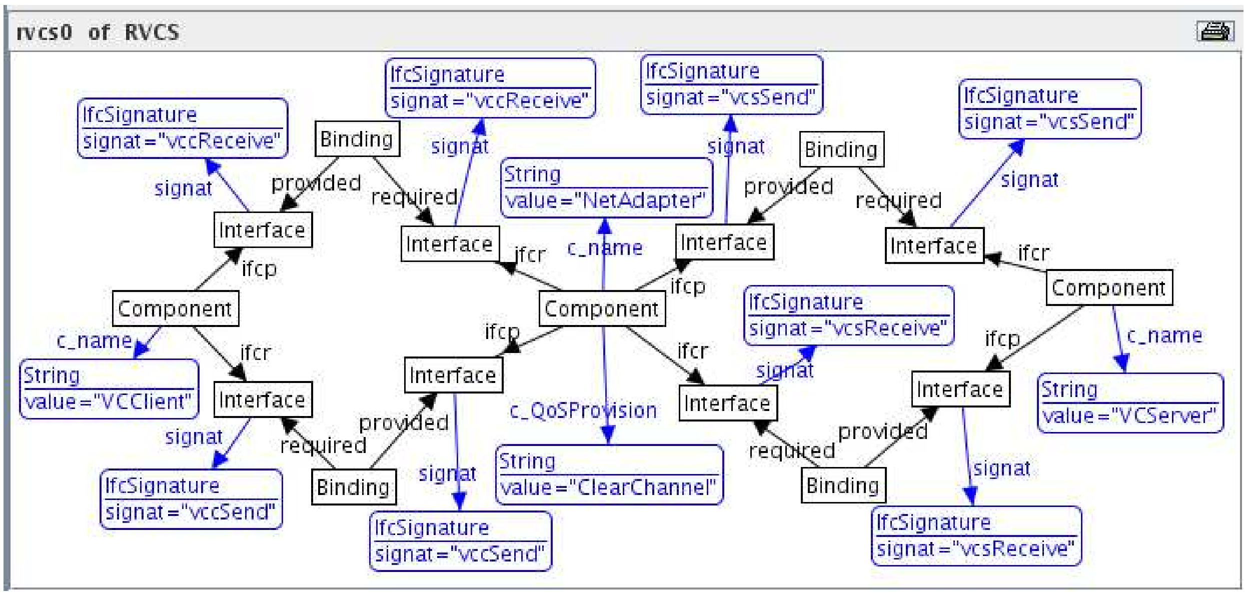

rvcs0 of RVCS
IfcSignature signat="vccReceive"
IfcSignature signat="vcsSend"
IfcSignature signat="vcsReceive"
IfcSignature signat="vccSend"
Binding
Interface
Component
String value="NetAdapter"
String value="VCClient"
String value="VCServer"
String value="ClearChannel"
signat
provided
required
ifcp
ifcr
c_name
c_QoSProvision

Figure 4.8: E-Graph CBS reflection structure for the RVCS example.

4.2.3 QoS Contracts Structure

Conceptually, a QoS contract specifies expected QoS levels for a given application service or functionality, under different context conditions. These QoS levels become obligations that can be seen as the guarantees offered by a system or service provider to its users [Beugnard *et al.*, 1999, Keller and Ludwig, 2003, Collet *et al.*, 2005, Krakowiak, 2009, Tran and Tsuji, 2009], whenever the respective context conditions occur at runtime. 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 conditions must be permanently monitored and the system must act upon their violation in order to have the possibility of restoring it opportunely.

For a software system to address the violation of its QoS contracts, it must incorporate and manage these contracts internally. Given that we use e-graphs to define the formal model of component-based system structure as a realization of system reflection, we follow the same idea to define QoS contracts as a manageable part of the software system.

Definition 4.5 (QoS Contract –QoSC). *Given $QoSDSig$ the usual data signature over the disjoint union $Integer + String + Boolean + Predicate + IntPair$, where $IntPair = Integer \times Integer$ with the usual interpretations, a QoS contract is a tuple (C, ct), where*

- *C is an e-graph representing a contract instance, subject to*
- *ct, an e-graph morphism $ct : C \to QoSC$, where $QoSC$ is the e-graph definition for typing QoS contracts, $QoSC = (V_1, V_2, E_1, E_2, E_3, (source_i, target_i)_{i=1,2,3})$ such that*
  - $V_1 = \{QoSContract, QoSProperty, QoSMonitor, QoSGuarantor, SLOObligation, QoSRuleSet\}$;
  - $V_2 = Integer + String + Boolean + Predicate + IntPair$;
  - $E_1 = \{property, obligation, monitor, guarantor, ruleSet\}$;
  - $E_2 = \{gname, pname, mname, rname, SLOPredicate, contextCondition, isActive, contextEvType, QoSCmult, QoSPmult, QoSMmult, QoSGmult, SLOOmult, QoSRmult\}$;
  - $E_3 = \{pmult, omult, mmult, gmult, rmult\}$;
  - $source_1 = \{(property, QoSContract), (obligation, QoSProperty), (monitor, QoSProperty), (guarantor, QoSProperty), (ruleSet, SLOObligation)\}$;
  - $target_1 = \{(property, QoSProperty), (obligation, SLOObligation), (monitor, QoSMonitor), (guarantor, QoSGuarantor), (ruleSet, QoSRuleSet)\}$;
  - $source_2 = \{(gname, QoSGuarantor), (pname, QoSProperty), (mname, QoSMonitor), (rname, QoSRuleSet), (SLOPredicate, SLOObligation), (contextCondition, SLOObligation), (isActive, SLOObligation), (contextEvType, SLOObligation), (QoSCmult, QoSContract), (QoSPmult, QoSProperty), (QoSMmult, QoSMonitor), (QoSGmult, QoSGuarantor), (SLOmult, SLOObligation), (QoSRmult, QoSRuleSet)\}$;
  - $target_2 = \{(gname, String), (pname, String), (mname, String), (rname, String), (SLOPredicate, Predicate), (contextCondition, Predicate), (isActive, Boolean), (contextEvType, String), (QoSCmult, Integer), (QoSPmult, Integer), (QoSMmult, Integer), (QoSGmult, Integer), (SLOmult, Integer), (QoSRmult, Integer)\}$;
  - $source_3 = \{(pmult, property), (omult, obligation), (mmult, monitor), (gmult, guarantor), (rmult, ruleSet)\}$; *and*
  - $target_3 = \{(pmult, IntPair), (omult, IntPair), (mmult, IntPair), (gmult, IntPair), (rmult, IntPair)\}$.

Our QoS contract definition comprises a set of *QoSPropert*ies. Each QoS property has a set of QoS-level obligations (*SLOObligation*). Each SLO obligation has (i) the QoS level (*SLOPredicate*) to be fulfilled under (ii) a corresponding context condition (*contextCondition*); (iii) a non-empty guaranteeing reconfiguration rule-set (*QoSRuleSet*) to apply upon notification of events that belong to (iv) the respective non-empty set of triggering context events (defining the type *contextEvType*); and (v) a flag to mark its status (*isActive*). Different QoS levels naturally imply disjunctive context conditions, and also disjunctive sets of triggering context events. The *QoSGuarantor* refers to a system element distinguished among those with the responsibility of fulfilling the SLO obligations, while the *QoSMonitor*, to the one that notifies the respective context events.

This definition involves the coherent assignment of responsibilities for the coordinated operation of the elements required to perform a software reconfiguration [Kephart and Chess, 2003]. That is, monitoring elements notify context events, signaling a new context condition (*contextCondition*). For this new condition, the contract specifies a respective QoS level to be fulfilled (*SLOObligation*) and a guaranteeing rule-set (*QoSRuleSet*). This rule-set is used to perform the reconfiguration of the component-based structure application representation (*CBSAR*) of the actual running system.

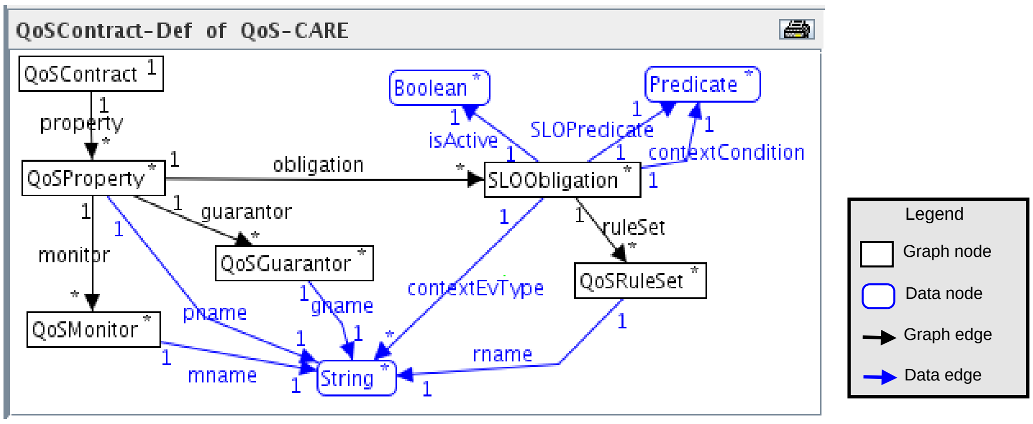


Figure 4.9: Illustration of our E-Graph based QoS contract definition. The *Integer* type, *mult* and *multiplicity* attributes, for multiplicity constraints, are presented implicitly in the usual (short) notation.

By using the same formalism to define QoS contracts as a manageable part of the system, QoS contracts themselves are enabled to be dynamically reconfigured, if necessary. This can be performed using the same mechanism used to reconfigure the managed application. However, for this to be automated, this would require additional mechanisms such as a second feedback-loop to perform the adaptation of these contracts, given the definition of higher-level goals, as defined in [Kramer and Magee, 2009] or [Villegas *et al.*, 2012].

Example 4.2 (QoS Contract on Confidentiality). *We previously presented the confidentiality requirements for the videoconference software system in our application scenario. These requirements are synthesized in table 4.2, which specifies the expected QoS levels to be guaranteed by this application under different context conditions, signaled by respective context events. It also specifies the guaranteeing reconfiguration rule-sets to be applied to fulfill the respective QoS levels. The first row (labeled 1) establishes that whenever the client connects from an intranet-serviced area (i.e., notified by "from_intranet" context events) she should expect a QoS level corresponding to a clear channel communication ("clearChannel"). The corresponding guaranteeing rule-set, defined by a software evolution architect once negotiated the QoS levels with the client, is named "R_clearChannel". The reconfiguration rules (as defined in the next section) in this rule-set must be given by the user in such a way that they address the specified QoS levels. The interpretation is similar for the other two rows.*

The NetAdapter component is specified as the QoSGuarantor, and an AccessPointProbe on this component as the QoSMonitor for the confidentiality QoS property. This monitor is used by the reconfiguration mechanism to continuously check the changes in the context conditions and violations of the actual QoS level. The e-graph representation for this contract is given in Fig. 4.10.

4.2.4 The Component-Based Structure Reconfiguration System

Having formalized the structural parts of a managed software application in terms of e-graphs, we define our software reconfiguration system by extending typed attributed graph transformation systems. Our reconfiguration system is supported by our definitions of component-based structure, application reflection, QoS contract, and the following on reconfiguration rules.

Definition 4.6 (CBS Reconfiguration Rule –CBSRR). *A component-based structure reconfiguration rule, $CBSRR$, is a tuple $(L, K, R, l, r, lt, kt, rt)$, abbreviated $CBSRR = (L \xleftarrow{l} K \xrightarrow{r} R)$, where L (left hand side), K (left-right gluing), and R (right hand side) are e-graphs related through graph morphisms l, r. $CBSRR$ denotes a graph-rewriting rule transforming L into R. The graph morphisms*

Table 4.2: QoS contract example on confidentiality for the RVCS application

Videoconference Application Obligations		
Context Events	Expected QoS Level	Guaranteeing Rule Set
1: $from\_intranet$	$clearChannel$	$R\_clearChannel$
2: $from\_extranet$	$confidentChannel$	$R\_confidentChannel$
3: $no\_network\_conn$	$localCache$	$R\_localCache$
Assignment of Responsibilities		
- System Guarantor:	$System.NetAdapter$[a]	
- Context Monitor:	$System.NetAdapter\_AccessPointProbe$[b]	

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

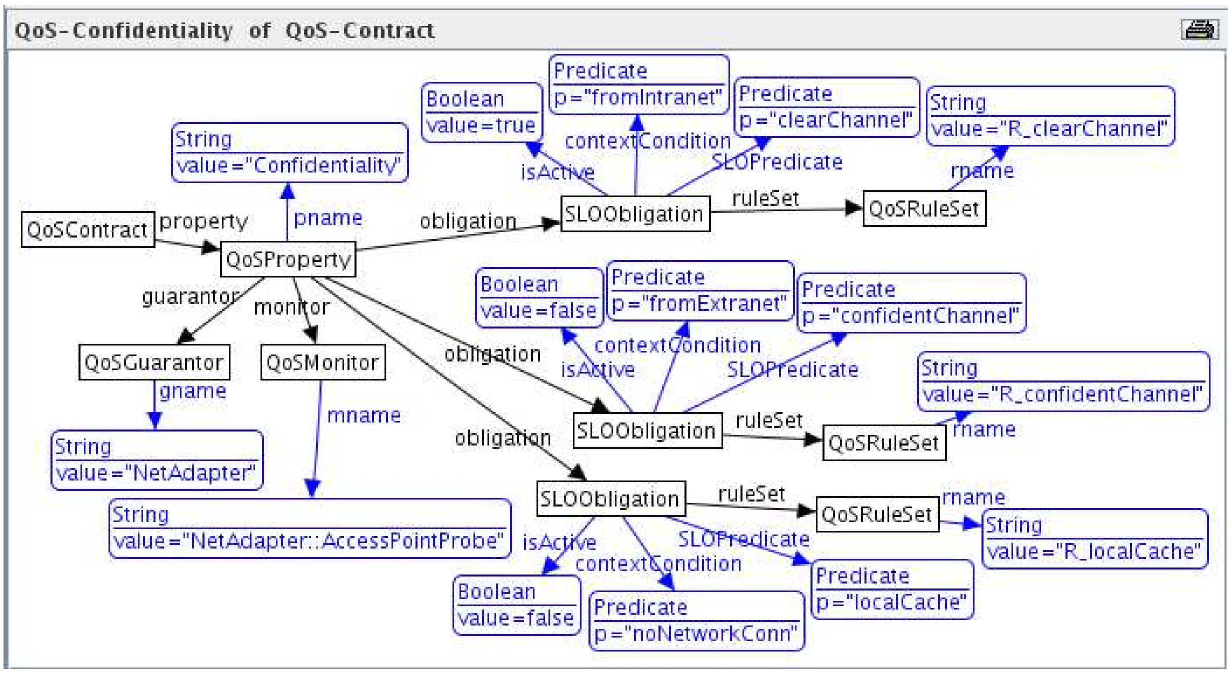

QoS-Confidentiality of QoS-Contract
QoSContract
property
QoSProperty
String value="Confidentiality"
pname
obligation
SLOObligation
Boolean value=true
isActive
Predicate p="fromIntranet"
contextCondition
Predicate p="clearChannel"
SLOPredicate
ruleSet
QoSRuleSet
rname
String value="R_clearChannel"
guarantor
monitor
QoSGuarantor
QoSMonitor
gname
mname
String value="NetAdapter"
String value="NetAdapter::AccessPointProbe"
Boolean value=false
Predicate p="fromExtranet"
Predicate p="confidentChannel"
String value="R_confidentChannel"
Predicate p="localCache"
Predicate p="noNetworkConn"
String value="R_localCache"

Figure 4.10: E-Graph QoS contract example on confidentiality for the RVCS.

$lt : L \to CBS$, $kt : K \to CBS$ and $rt : R \to CBS$ ensure that the rule elements involve only component-based structures. $CBSRR$ rules are said to reconfigure L into R.

Conceptually, a reconfiguration rule encodes a strategy to address a particular QoS level on a QoS property. This kind of strategies have been analyzed and proposed in different computer science knowledge areas, having been encoded in *design patterns*. For example, Ramachandran and Dougherty *et al.* present comprehensive catalogs of *security* design-patterns at the architectural, design and implementation levels for enabling *valid* communications in transparent ways; all otherwise invalid communication, whether unauthorized, unauthenticated, unexpected, uninvited, or unwanted being blocked [Ramachandran, 2002, Dougherty *et al.*, 2009]. Krakowiak and Buschmann *et al.*, present *availability* and *performance* design-patterns for distributed systems [Krakowiak, 2009, Buschmann *et al.*, 2007]. Hence, system evolution architects can take advantage of known design patterns that address this kind of conditions on QoS properties to encode them in the left and right hand sides of reconfiguration rules. All left-hand sides of rules in a rule-set are named after that rule-set name. Different left hand sides for a similar right hand side in a rule-set are possible as far as they have no termination conflicts, as we show in Section 6.4).

Example 4.3 (Reconfiguration rule)**.** *The QoS contract on confidentiality for our videoconference example specifies a guaranteeing set of reconfiguration rules, R_confidentChannel, to be applied when the user moves to an extranet-serviced area and the contract is in risk of being violated. Figure 4.11 illustrates the rule, in that set, that applies when the user is moving from the intranet. The left-hand side (LHS) of the rule is used by pattern-matching to find a component that supports a ClearChannel (by its c.QoSProvision attribute). The right-hand side (RHS) specifies that (i) the matched components by the LHS must be kept with their corresponding interface bindings, except those identified with indexes* 4 *and* 5 *("4:Binding" and "5:Binding"); (ii) the new (highlighted) elements must be configured and deployed to provide a tunneled (i.e., confident) channel; (iii) the new leftmost interfaces must be connected to the previously existing ones, indexed with* 2 *and* 3*, and those indexed with* 4 *and* 5 *must be reconnected to the new rightmost interfaces, respectively; and (iv) the c_QoSProvision attribute must be updated as provisioning a SecureChannel. For readability reasons, 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-highlighted elements in the RHS with their LHS counterparts.*

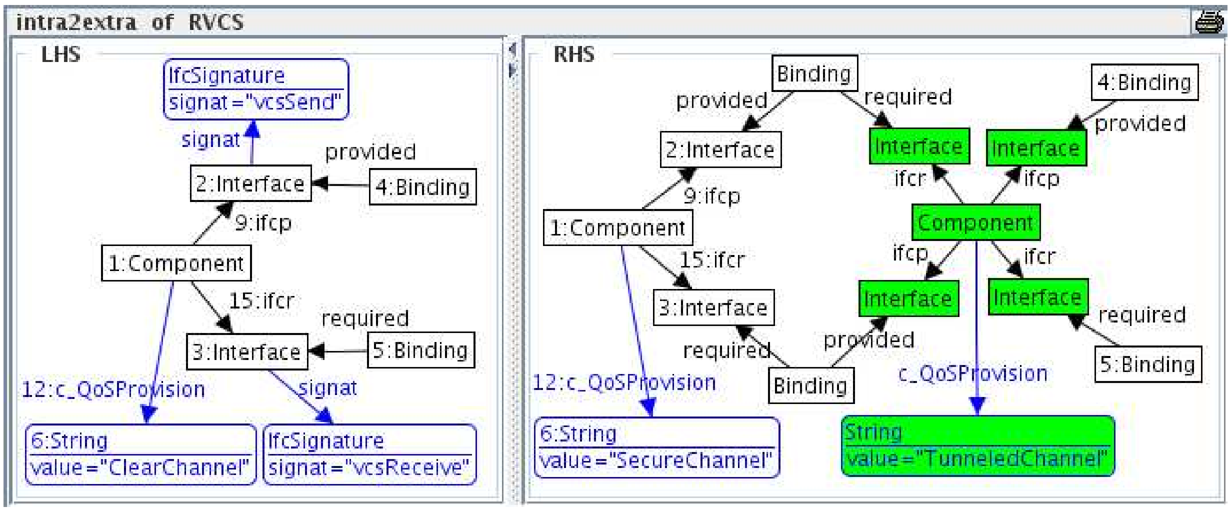


Figure 4.11: The E-Graph $R\_confidentChannel$ reconfiguration rule.

Negative Application Conditions. Left-hand sides (LHSs) of reconfiguration rules are called "application conditions" as they specify the necessary conditions that a managed application structure must fulfill for a rule to be applied. Fulfilling these conditions means to find a match of the LHS in the system structure. However, reconfiguration rules can (optionally) have one or more *negative application conditions (NACs)* to specify conditions that avoid their application. Each NAC is specified in a rule $(L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ by associating its LHS L with an e-graph N through a respective graph morphism $n : L \rightarrow N$. To apply a rule with NACs, no match must exist of its NACs in the system structure. That is, a NAC is satisfied through a match $m : L \rightarrow G$ if it is not possible to find a total morphism $t : N \rightarrow G$ such that any two graph objects mapped to one another by u are mapped to the same object in the system graph G. More formally, there must exist no total morphism $t : N \rightarrow G$ such that $t \circ u = m$.

Definition 4.7 (CBS Reconfiguration System –CBSRS)**.** *The component-based structure reconfiguration system, CBSRS, is a tuple $(DSig, CBS, \Re_S, P)$, where $DSig$ is the data type signature from Def. 4.3 (CBS); CBS the component-based structure definition; $\Re_S$ the reflection structure of the managed application to reconfigure in its actual running state; and P a set of reconfiguration rules, with CBS, $\Re_S$, and P according to Def. 4.3, 4.4, and 4.6, respectively, for:*

1. *(What and Where to reconfigure) The rule-set P is applied to the managed application reflection structure $\Re_S = (G, f_S, t)$ by graph-based pattern matching. That is, for each reconfiguration rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in P, we determine whether there exists a match of p in $\Re_S$. For this, we define the boolean $match(p, \Re_S)$ function, which returns True if there exists a morphism $m : L \rightarrow G$ (called a matching of the left-hand side of p, L, in G), and False otherwise. This matching determines the CBS potential elements to be reconfigured.*

2. *(How to reconfigure) We call a direct reconfiguration $G \overset{p,m}{\Rightarrow} H$ the application of the e-graph transformation of G into H, as specified by the reconfiguration rule p, according to Def. 4.6.*

3. *(One-step CBS reconfiguration) Finally, the CBS reconfiguration is performed through the $reconfig(\Re_S, P)$ operation. That is, the sequence of direct transformations $\Re_S = G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n = \Re'_S$, written $\Re_S \overset{*}{\Rightarrow} \Re'_S$, is performed until no more rules in P can be applied, obtaining a new managed application reflection structure, $\Re'_S$. We guarantee that this sequence is terminating at configuration time by checking the parameterized reconfiguration rules for termination conflicts, as we show in Section 6.4. From the implied operations in the sequence of transformations, a whole, correct-by-construction reconfiguration plan is synthesized.*

Example 4.4 (Managed application reconfiguration)**.** *After applying the reconfiguration rule of Example 4.3, we obtain the reconfigured application structure illustrated in Fig. 4.12. This new structure fulfills the QoS level (i.e., confidentChannel) for the new context condition (i.e., network connection fromExtranet), as specified in the contract. In this figure, the components that were added and deployed by the applied reconfiguration rule are highlighted. Correspondingly, Fig. 4.13 illustrates the reconfigured runtime application structure in component-based notation.*

Finally, it is worth noting in our reconfiguration system (CBSRS) that our QoS Contract definition involves all the required elements to address the questions *when*, *what*, *where* and *how* to reconfigure a managed software application to preserve its contracts. In this section we answered in detail the latter three questions. In the next section we precise our answer to the *when* question by formally modeling the semantics of our QoS contract definition using finite-state machines. As we introduced previously, the reconfiguration cycles, whose goal is the preservation of QoS contracts, start and end in actual running managed software applications, triggered by context events.

4.3 Finite State Machine Modeling of QoS Contracts States

A Finite State Machine (FSM) is specified by a set of states and transitions among them, being useful for modeling the global operation of a software system in terms of its states with respect to a particular aspect of interest [Hopcroft *et al.*, 2006]. We use FSMs to (i) model the QoS contracts states and transitions, not only for the (QoS levels) states of contract fulfillment, but also for the states of contract *unfulfillment*; (ii) maintain the consistent association of these contract states with running-software states; and (iii) control QoS contract-preserving reconfiguration cycles according to context changes. In this way, FSMs add to our model robustness with respect to context unpredictability.

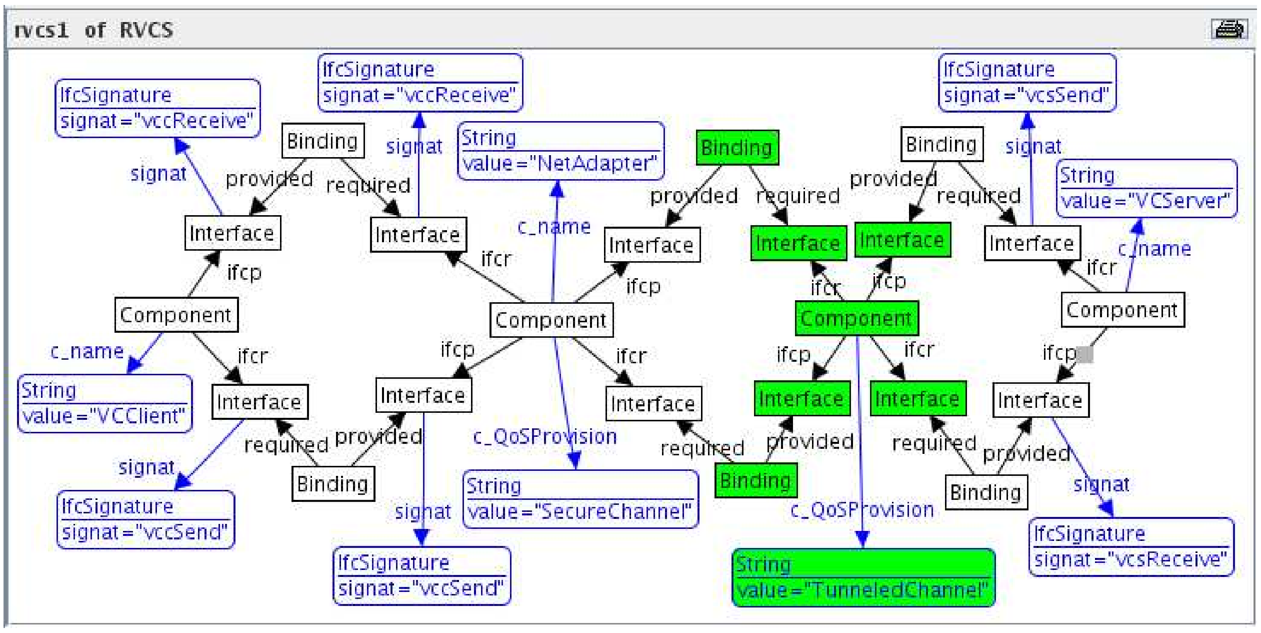

rvcs1 of RVCS
IfcSignature signat="vccReceive"
IfcSignature signat="vccReceive"
IfcSignature signat="vcsSend"
String value="NetAdapter"
String value="VCServer"
String value="VCClient"
IfcSignature signat="vccSend"
IfcSignature signat="vccSend"
String value="SecureChannel"
String value="TunneledChannel"
IfcSignature signat="vcsReceive"
Binding
Interface
Component
signat
provided
required
ifcp
ifcr
c_name
c_QoSProvision

Figure 4.12: E-Graph reconfigured reflection structure for the RVCS example.

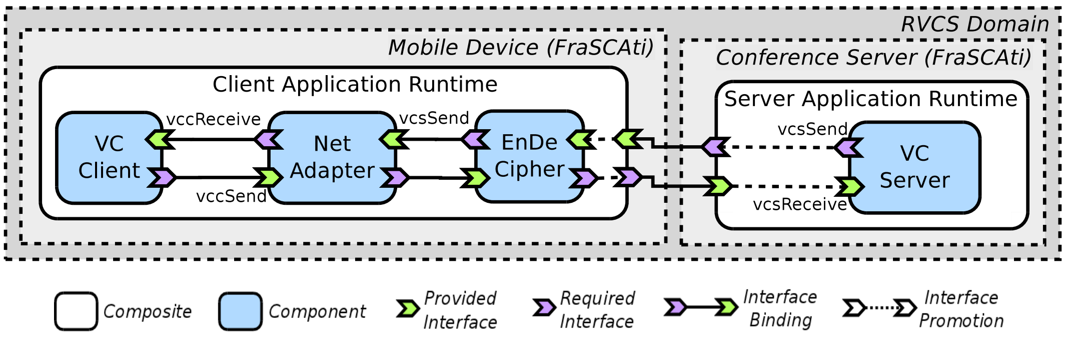


Figure 4.13: Reconfigured application structure for the RVCS example.

4.3.1 An Initial Interpretation of QoS Contracts as FSMs

Our definition of QoS contract (*QoSC*, Def. 4.5 on page 72) comprises a set of QoS properties (through its *property* attribute). Each *property* has a set of obligations (*obligation*), where each *obligation* specifies:

- a QoS-level objective (*SLOPredicate*) to be fulfilled under
- a specific context condition (i.e., *contextCondition*), whose occurrence is signaled by
- a type for the context events that notify of changes to this context condition (i.e., *contextEvType*); and
- a set of guaranteeing reconfiguration rules (i.e., *ruleSet*) to reconfigure the managed application, with the expected effect of fulfilling the corresponding QoS level.

On the other side, an FSM is a 5-tuple $\langle STATES, start, ACCEPT, \Sigma, \delta \rangle$, where

- $STATES$ is a finite set of *states* (the QoS levels specified in the contract),
- $start \in STATES$ is the contract state that corresponds to the managed software applica-

tion state when initially executed,

- $ACCEPT \subseteq STATES$ is a set of *accepting states* (in our case $ACCEPT = STATES$, given that all contract states are acceptable finishing states for the execution of managed software applications),
- Σ is a finite *input alphabet* (the set of context events to signal context changes, thus implying changes of QoS levels), and
- $\delta : STATES \times \Sigma \rightarrow STATES$ is the *transition function*.

Thus, from the relationship between these two structures, *QoSC*s and FSMs, it is easy to obtain Algorithm 4.1 to map a QoS contract into the corresponding FSM.

Algorithm 4.1 FSM_from_QoS_Contract_First_Version

```
Input: $qosContract : QoSC$ /* QoSC from Def. 4.5 */
Output: $fsm : tuple\langle STATES, start, ACCEPT, \Sigma, \delta\rangle$
 1: $fsm.STATES \leftarrow \{\}$
 2: $fsm.\Sigma \leftarrow \{\}$
 3: $fsm.\delta \leftarrow \{\}$
 4: for all $qosProp \in qosContract.C.property$ do
 5:    for all $sloOblig \in qosProp.obligation$ do
 6:       $curstate \leftarrow make\_indexed\_state(sloOblig.SLOPredicate)$
 7:       $fsm.STATES \leftarrow fsm.STATES \cup \{curstate\}$/* build all the FSM states */
 8:       $fsm.\Sigma \leftarrow fsm.\Sigma \cup sloOblig.contextEvType$ /* and the FSM events */
 9:    end for
       /* Then define the FSM transitions among the states: */
10:    for all $sloOblig \in qosProp.obligation$ do
11:       $curstate \leftarrow get\_indexed\_state(fsm.STATES, sloOblig.SLOPredicate)$
12:       for all $event \in sloOblig.contextEvType$ do
13:          for all $eachstate \in fsm.STATES\}$ do /* Including the $curstate$ */
14:             $fsm.\delta \leftarrow fsm.\delta \cup \{(\langle eachstate, event\rangle, curstate)\}$
15:          end for
16:       end for
17:    end for
18: end for
19: $fsm.ACCEPT \leftarrow fsm.STATES$
20: return $fsm$
```

This algorithm produces correct-by-construction FSMs with respect to the QoS contract specification. However, we can distinguish between two different kinds of context conditions that can be included in QoS contracts: (i) context conditions that may occur freely, independent of any other context conditions; and (ii) context conditions whose occurrences depend on the occurrence of other context conditions. We present instances of the first case in Example 4.5, while the cases corresponding to the second case are analyzed in the next section.

Example 4.5 (Straight FSM representation of QoS contracts)**.** *Applying Algorithm 4.1 to the contract example on confidentiality illustrated previously, we obtain the FSM presented in Fig. 4.14. In this figure, states represent QoS levels to be fulfilled by the videoconference application (i.e., clear channel structure, corresponding to the application state to be configured when connected from the intranet; confident channel structure, to be configured when connected from the extranet; and local cache structure when having no network connection). On each state, monitors notify context events correspondingly of connection types fromIntranet, fromExtranet, and noNetwork. It is worth noting the correspondence between the QoS levels specified in the contract and the FSM states, as well as how the transitions express the possibilities for the user to move indistinctly from any place to another at anytime, being these* *places either in intranet or extranet-serviced areas, or even having no access at all to any network. That is, the occurrence of any of these conditions are independent of the occurrences of the others. We observe a similar situation for the FSM representation of a contract on availability illustrated in Fig. 4.15. In this case, monitors notify context events on bandwidth (bw) changes (i.e., events correspondingly of types bw_lte_12 (i.e., $bw \leq 12$), bw_between_12_128 (i.e. $12 < bw < 128$) and bw_gte_128 (i.e. $bw \geq 128$)).*

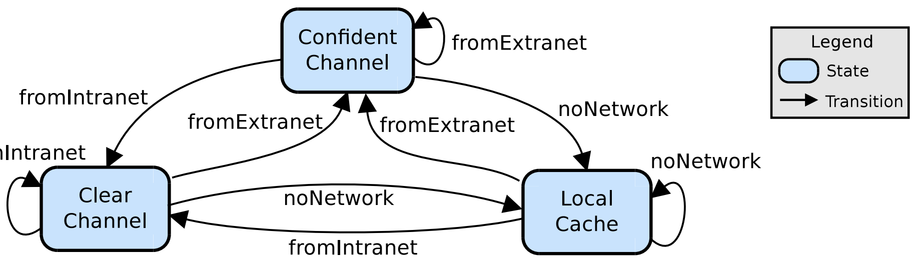


Figure 4.14: Finite state machine representation for the QoS contract example on confidentiality.

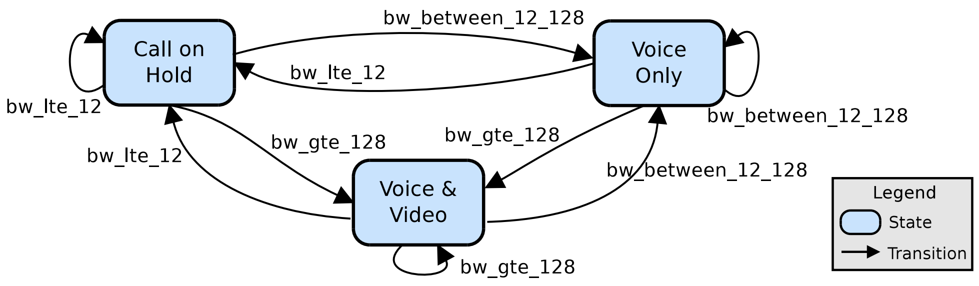


Figure 4.15: Finite state machine representation for a QoS contract on availability.

4.3.2 Reformulation of FSMs to Model QoS Contracts States

In this section we illustrate and analyze the cases corresponding to the QoS context conditions whose occurrences depend on the occurrence of other context conditions. The following example illustrates one of such cases.

Example 4.6 (FSM representation of a QoS contract on throughput)**.** *Table 4.3 illustrates a contract example for software services whose minimum throughput rates depend on calendar dates, such as the monthly payment of taxes on sales*[26]*. Given that people typically pay these services in the last (increasingly) few days of the month, the software applications implementing these services must support correspondingly the expected high increments in service load for these days. Thus, for the first 15 days of the month (row 1 of the table), the expected throughput is of 100 transactions per minute, whereas for the last 4 days it is of 900 transactions per minute (row 4). Figure 4.16 presents the corresponding FSM obtained with our algorithm.*

It is worth noting that for the cases of context conditions with inter-dependencies, our algorithm produces superfluous transitions. For instance, in the first state "T100/min" the transition corresponding to the event of changing to day 24—*day24* labeled (A) in Fig. 4.16—is clearly superfluous. In this state (i.e., between days 1 and 15), the event changing to day 24 will not occur before the occurrence of events changing to days 16 to 23, which are already specified by their

[26]For simplicity reasons, we only consider the conditions and transitions for 31-day months.

Table 4.3: QoS contract example on throughput

Videoconference Application Obligations		
Context Events	Service Level Objective	Guaranteeing Rule Set
1: $day1, day2\_15$	$T100/min$	$R\_baseConfig$
2: $day16, day17\_23$	$T150/min$	$R\_1.5xbaseConfig$
3: $day24, day25\_27$	$T300/min$	$R\_3xbaseConfig$
4: $day28, day29\_31$	$T900/min$	$R\_9xbaseConfig$
Assignment of Responsibilities		
- System Guarantor:	$System.TaxProcessor$[a]	
- Context Monitor:	$System.OSService.DateTimeProbe$[b]	

[a] The software application component responsible for processing tax requests, provisioned with capabilities for supporting scalable throughput.
[b] The designated component for monitoring changes on calendar dates.

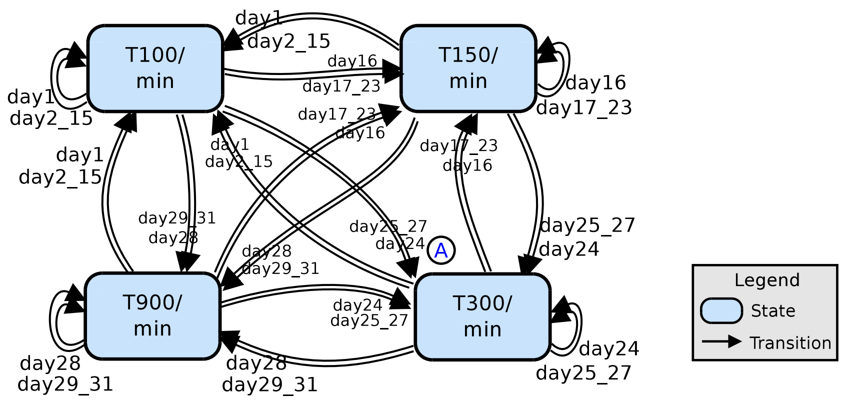


Figure 4.16: Finite state machine obtained applying Algorithm 4.1 to the contract example on throughput.

respective transitions. In contrast, Fig. 4.17 presents the FSM version without such superfluous transitions. These superfluous transitions are avoided by considering the dependencies among context conditions. Thus, these dependencies determine the transitions among states.

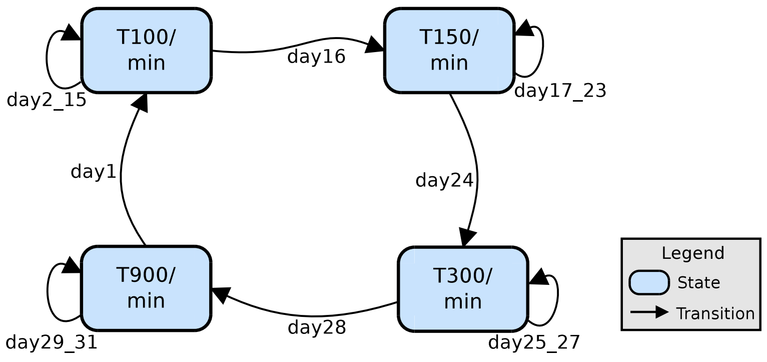


Figure 4.17: A second FSM for the QoS contract on throughput, without superfluous transitions.

Moreover, from a more detailed examination of the definition of FSMs and our Algorithm 4.1, we identify other problems worth of improving the FSM realization of our QoS contracts.

First, in this initial interpretation, 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 [Harel, 1987] yet, in some cases, superfluous transitions. Second, the reconfiguration rules are not considered in the FSMs, thus a different invocation mechanism is needed for their execution in the FSM. Even modifying the FSM to support entry/exit actions as in the event-condition-action strategies [Colombo *et al.*, 2006], the application of our reconfiguration rule *sets* would require additional mechanisms, as event-condition-action strategies specify only one action per event-condition. Third, the objective of QoS contracts is to specify expected QoS levels on software services, thus they specify only states of *fulfillment* with their corresponding transitions. Hence, 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). More importantly, in light of this required robustness, the superfluous transitions generated by our algorithm become possible sources of additional unexpected and undesirable states, which should be avoided.

Nonetheless, realizing these improvements 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 these states, which is of course contradictory to the problem of transitions explosion. Yet another solution could be to use "wild card" events. However, this would also imply that our algorithm would have to generate transitions from every state of contract fulfillment to each of the undesirable states, with the added difficulty of specifying the semantics of each wild card event in the transitions.

Therefore, our solution is to take advantage of our component-based reconfiguration system based on e-graphs ($CBSRS$, Def. 4.7 on page 76) that we defined previously, as a more powerful way of specifying the FSM transitions. For this, we (i) parameterize the FSM with our $CBSRS$, (ii) add five auxiliary functions, and (iii) replace the transition function (δ) of the original FSM definition, as follows.

Definition 4.8 (QoS-Contract FSM –*QoSC_FSM*). *A QoSC_FSM is the tuple* $\langle STATES, ACCEPT, \Sigma, \Gamma, \Psi, \kappa, \eta, \rho, \pi, CBSRS, \delta\rangle$, *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. 4.6). 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. $\mathcal{P}(\Gamma)$ is the power set of Γ.*
- *$\pi : STATES \rightarrow QoSProperty$ 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. 4.7, p. 76).*
- *$\delta : CBSAR \times \Sigma \times \Delta^U \rightarrow STATES$ is the transition function, where $CBSAR$ is the Component-Based Structure Application Reflection definition (Def. 4.4), and Δ^U is a reference to the universal execution context space of software applications. That is, we leave this universal context space intentionally unspecified in this definition, given the practical impossibility of formalizing it, even in an abstract form. Any abstraction would imply a partial representation of the context of execution of any software system, which, at some point would turn to be incomplete and formally unsound. Instead, having it as a reference to an abstract and external construct allows us to refer to its particular instances when evaluating the reconfigured managed application by sensing, directly or indirectly, the actual managed application's context of execution, which is more realistic and feasible than formalizing it.*

In the following sections we illustrate how this new definition of state machine, in the context of our e-graph based QoS-driven reconfiguration model, solves the aforementioned problems, including the management of unexpected and undesirable states. The inclusion of these states accounts to address robustness, according to the goals of this dissertation.

4.3.3 The Exception and Unstable States of Contract Unfulfillment

Our previous interpretation of QoS contracts is based on the QoS-level objectives to fulfill, meaning that up to this point our model considers only the states of contract fulfillment. To address robustness facing situations unforeseen by the user and derived from unexpected context situations, we identify two states of contract *unfulfillment*. These states, to be added to the states of contract fulfillment in their automatic derivation from QoS contracts, 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 rule(s) 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. 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.

4.3.4 The QoSC_FSM Semantics

So far, we have given a new definition of FSMs—QoSC_FSMs—tailored for our model of QoS contract-driven and e-graph based reconfiguration system. Based on this definition and our previous considerations, we refine our previous algorithm to obtain Algorithm 4.2, which adds the states of contract unfulfillment to the set of states, and defines the mappings for the added auxiliary functions. This algorithm produces correct-by-construction FSMs with respect to QoS contract specifications considering conditions with and without inter-dependencies. However, the mapping between QoS contracts and *QoSC_FSMs* given by our new algorithm constitutes only the structural part of the QoSC_FSM semantics. Thus, in this section we complete the dynamic part of this semantics by defining δ, the transition function.

Algorithm 4.2 FSM_from_QoS_Contract_Definitive_Version

```
Input: $qosContract : QoSC$ /* QoSC from Def. 4.5 */
Output: $qfsm : \langle STATES, start, ACCEPT, \Sigma, \Gamma, \Psi, \kappa, \eta, \rho, \pi, CBSRS \rangle$ /* QoSC_FSM, Def. 4.8 */
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: $exceptionState \leftarrow make\_indexed\_state(Exception)$
19: $qfsm.STATES \leftarrow qfsm.STATES \cup \{exceptionState\}$
20: $unstableState \leftarrow make\_indexed\_state(Unstable)$
21: $qfsm.STATES \leftarrow qfsm.STATES \cup \{unstableState\}$
22: return $qfsm$
```

Intuitive Semantics

To informally illustrate the semantics of QoSC_FSM states and transitions we use the contract example on confidentiality (cf. Example 4.2 on page 73), specifically when the videoconference system receives the event of entering into an extranet-serviced area.

As mentioned before, a QoSC_FSM controls the QoS levels in its contracted states of both, fulfillment and unfulfillment. However, we must emphasize that the final objective of QoSC_FSMs is to maintain, as far as possible, the managed software application in the states that correspond to those of QoS contract fulfillment. This is a major responsibility of the transition function, of course, supported by the auxiliary functions that we included in our redefined FSM, and specially the reconfiguration rules given by the user. Hence, the states of fulfillment to be reached acquire a predominant importance in the definition of our transition function. That is, *independently* of the current state of the managed software application, what is most important in a transition triggered by a new context condition is that the managed software application *ends* fulfilling the QoS-level objective that corresponds to the new condition. However, if reaching a fulfillment state is not possible, our model determines the respective causes to notify the user, and leaves the system in a corresponding state of unfulfillment, as explained before.

In light of this, we recall that our component-based structure reconfiguration system, *CBSRS* from Def. 4.7, specifies precisely *how* to reach a fulfillment state from any of the possible managed application states, using graph-based pattern matching and graph transformation with the rules given by the user. This is achieved through the guaranteeing rule-set associated to each of the QoSC_FSM states. All rules in a rule-set are specified in left-hand and right-hand sides (LHS and RHS respectively), according to Def. 4.6, as follows. On one side, the RHSs of all rules in a same rule-set encodes a design pattern (possibly the same) to be "implanted" in the target system to fulfill the expected QoS level. On another side, corresponding LHSs encode the specific patterns that represent the different relevant source states that can present the managed application when it is notified with context changes.

In the case of our example on confidentiality, entering into an extranet-serviced area implies the corresponding QoS level to be fulfilled, thus requiring a confidential channel. Additionally, the contract specifies two other possible context conditions entailing different QoS levels: having a network access from an intranet, and having no network access at all. Hence, the guaranteeing rule-set associated to this QoS level must have two rules. The LHS of the first rule encodes the minimum pattern to be identified in the state of the managed application when it is connected from the intranet, that is, having a clear channel network connection. This pattern specifies the minimum characterizing elements to identify it in a managed application structure, which are in this case the components responsible for maintaining the network connection and the corresponding *c_QoSProvision* attribute, as we illustrated previously (cf. Fig. 4.11). Correspondingly, the LHS of the second rule encodes the pattern representing the state of the managed application when having no network access, that is, having a local cache. The RHS of these two rules encodes a pattern to deploy and interconnect components that encipher/decipher the transmitted data, thus implementing a confidential channel and fulfilling the target QoS level, as illustrated in the same figure.

QoSC_FSM Added Functions

To support the definition of our δ transition function we added five auxiliary functions to the QoSC_FSM structure: Ψ, κ, η, ρ and π. 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; $\Psi : STATES \rightarrow PREDICATE$ 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 reconfiguration rule-set to fulfill the QoS level associated to a given state. Concerning π and κ, as we use them for managing multiple QoS properties in a QoS contract, and performing operational verification of context conditions in the formal model implementation, we defer their explanation to Sections 4.3.5 and 5.4.1, respectively.

The mappings for Ψ, κ, η, ρ, and π are automatically obtained from QoS contract specifications, resulting from lines 2 to 16 of our Algorithm 4.2. For the example of the contract on confidentiality, corresponding to the QoS level to be achieved when moving to an extranet-serviced area, the mappings are the following[27]:

- $\Psi(clearChannel) \mapsto "clearChannel"$

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

- $\eta(clearChannel) \mapsto from\_intranet$, with $from\_intranet = \{FromIntranet\}$
- $\rho(clearChannel) \mapsto R\_clearChannel$
- $\Psi(confidentChannel) \mapsto "confidentChannel"$
- $\eta(confidentChannel) \mapsto from\_extranet$, with $from\_extranet = \{FromExtranet\}$
- $\rho(confidentChannel) \mapsto R\_confidentChannel$
- $\Psi(localCache) \mapsto "localCache"$
- $\eta(localCache) \mapsto no\_network\_conn$, with $no\_network\_conn = \{NoNetwork\}$
- $\rho(localCache) \mapsto R\_localCache$

Here we present the values returned by Ψ, which represent predicates, as strings. These predicates are evaluated in the execution of the reconfigured managed application under the actual 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 previous example on throughput (cf. Example 4.6), $\Psi(T100/min)$ is mapped to $"T100/min"$, which would be evaluated by sensing if the predicate "T100/min" is satisfied or not (i.e., the application is performing 100 transactions per minute at the invocation time). This function is fundamental in our model, given that it is not obvious that these conditions will hold after performing a reconfiguration.

In the context of our example, assume that we have a QoSC_FSM with the functions defined as presented above, obtained with our Algorithm 4.2. We also would 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. Then, in this situation we have:

- $e \in \eta(confidentChannel)$ holds, given $\eta(confidentChannel) \mapsto from\_extranet$, $from\_extranet = \{FromExtranet\}$, and $e = FromExtranet$.
- The next target state s corresponding to the context condition signaled by event e is given by $\exists s : s \in STATES \,|\, e \in \eta(s)$, that is, $s = confidentChannel$.
- The QoS level to fulfill in this target state s is $\Psi(\exists s : s \in STATES \,|\, e \in \eta(s))$, that is, $\Psi(confidentChannel) \mapsto "confidentChannel"$.
- The rule-set to apply to reach this target state s is $\rho(\exists s : s \in STATES \,|\, e \in \eta(s))$, that is, $\rho(confidentChannel) \mapsto R\_confidentChannel$.

Generalized Conditions for the Transitions to the QoSC_FSM Target States

Based on the previous definitions and the explanation of the QoSC_FSM structure, we can notably generalize the transition conditions to the states of contract fulfillment. For this, besides *evalInContext*, we recall the *CBSRS* boolean functions *match* and *reconfig* that we defined with it. $match(r, \Re_S)$ returns true if there exists a match of rule r in $\Re_S$, and false otherwise. Having a reconfiguration rule-set γ such that $r \in \gamma$ and $match(r, \Re_S)$, $reconfig(\Re_S, \gamma)$ uses γ to reconfigure $\Re_S$, a managed application reflection structure. If the reconfiguration succeeds, that is, 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 the transitions to contract fulfillment states, given the context event e, on the running-software application reflection state $\Re_S$, and under the execution context Δ, is:

$$\begin{gathered} e \in \Sigma \,\bar{\wedge}\, \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \bar{\wedge} \\ reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \,\bar{\wedge}\, evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F \big] \end{gathered} \tag{4.1}$$

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
- there exists 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 target state s (i.e., $r \in \rho(s \,|\, e \in \eta(s))$) such that we can find a match of r in the current graph structure of the managed application $\Re_S$),
- and after successfully reconfiguring the managed application reflection structure with the respective rule-set ($reconfig(\Re_S, \rho(s \,|\, e \in \eta(s)))$), the evaluation of the QoS level to be fulfilled in the target state is F (Fulfilled), that is, $evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F$.

It is worth noting that the existence of matching rules in the managed application upon a change in context conditions necessarily implies a reconfiguration. We use the symbols $\bar{\wedge}$ and $\underline{\vee}$ for the *consecutive logical* conjunction and disjunction, respectively. Consecutive expressions operated with the $\bar{\wedge}$ ($\underline{\vee}$) 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 [Aho and Ullman, 1972].

In condition (4.1) we recall the predominance of *target states* over source states that we emphasized previously. Thus, for each of the target states, which correspond to the contractual 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, as previously explained in the informal semantics presentation, in the generalized condition (4.1) 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).

Moreover, we follow the same strategy for the aforementioned added states of contract unfulfillment (*Unstable* and *Exception*). For the transitions to 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 Sect. 4.3.3. Hence, the condition to reach the *unstable* state is the same as (4.1), 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:

$$\begin{gathered} e \in \Sigma \,\bar{\wedge}\, \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \bar{\wedge} \\ reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \,\bar{\wedge}\, evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = U \big] \end{gathered} \tag{4.2}$$

For the *exception* state we have three disjoint conditions. The first expresses that the user gave an incomplete set of reconfiguration rules (i.e., no rule matches a given application state or context condition: $\forall r : r \in \rho(s \,|\, e \in \eta(s)) \big[\neg match(r, \Re_S) \big]$); 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., $\neg reconfig(\Re_S, \rho(s \,|\, e \in \eta(s)))$):

$$\begin{aligned}&e \in \Sigma \barwedge \exists s : s \in STATES \,|\, \forall r : r \in \rho(s \,|\, e \in \eta(s)) \big[\neg match(r, \Re_S)\big] \vee \\ &(e \in \Sigma^U \setminus \Sigma) \vee \\ &\big(e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge \\ &\neg reconfig(\Re_S, \rho(s \,|\, e \in \eta(s)))\big]\big)\end{aligned} \tag{4.3}$$

Definition of the Transition Function

From the previous illustration, it is easy to observe that the same condition (4.1) expresses correctly the transition requirements that target the three states of contract fulfillment of the confidentiality example, as illustrated in Fig. 4.18. In this figure, the output state of δ, that is, $\delta(\Re_S, e, \Delta)$, depends on the actual managed application reflection state ($\Re_S : CBSAR$), the context event received ($e \in \Sigma$), and the context of execution (Δ).

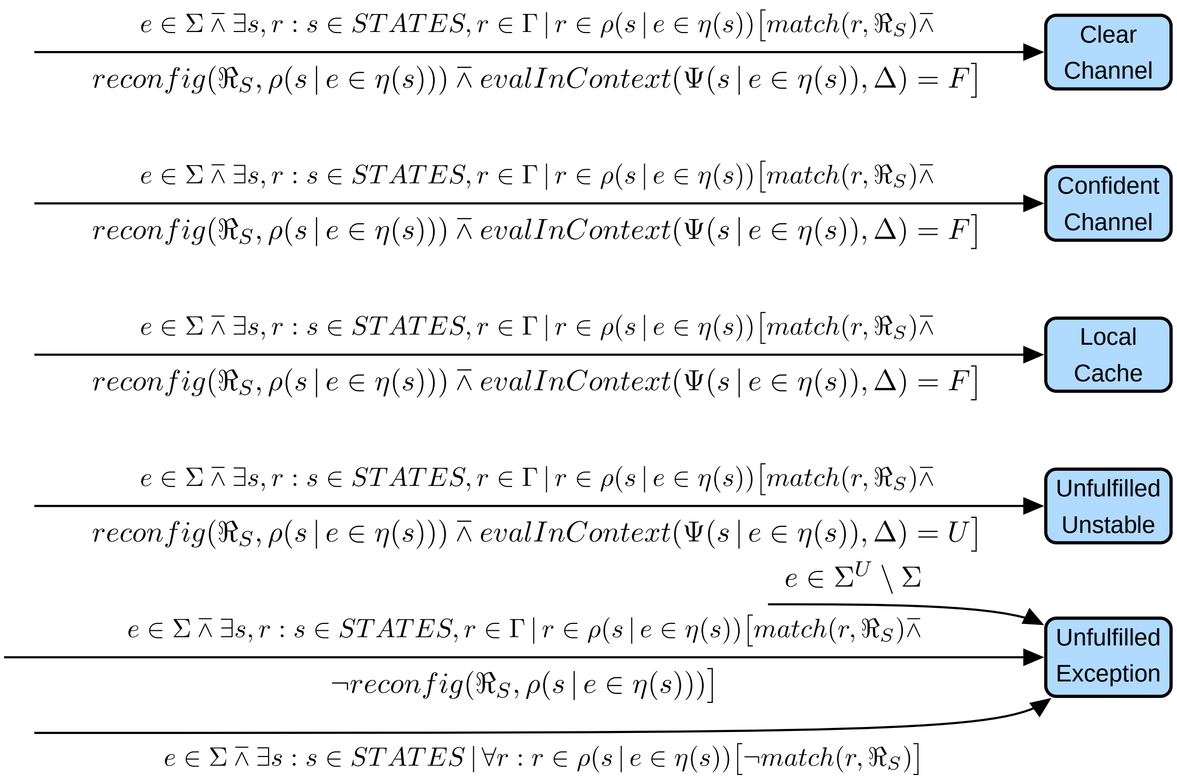

$e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge$
$reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F\big]$
Clear Channel
$e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge$
$reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F\big]$
Confident Channel
$e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge$
$reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = F\big]$
Local Cache
$e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge$
$reconfig(\Re_S, \rho(s \,|\, e \in \eta(s))) \barwedge evalInContext(\Psi(s \,|\, e \in \eta(s)), \Delta) = U\big]$
Unfulfilled Unstable
$e \in \Sigma^U \setminus \Sigma$
$e \in \Sigma \barwedge \exists s, r : s \in STATES, r \in \Gamma \,|\, r \in \rho(s \,|\, e \in \eta(s)) \big[match(r, \Re_S) \barwedge$
$\neg reconfig(\Re_S, \rho(s \,|\, e \in \eta(s)))\big]$
Unfulfilled Exception
$e \in \Sigma \barwedge \exists s : s \in STATES \,|\, \forall r : r \in \rho(s \,|\, e \in \eta(s)) \big[\neg match(r, \Re_S)\big]$

Figure 4.18: The QoSC_FSM transition function illustrated for the contract on confidentiality.

Based on this, we can abstract all the FSM fulfillment states as one "Contract Fulfillment" state to simplify the presentation of QoSC_FSMs without loss of generality[28]. In this way, our model of QoS contract preservation becomes the state-machine depicted in Fig. 4.19. In this figure, three possible transitions, corresponding to the generalized transition conditions (4.1), (4.2) and (4.3), may occur: (A) RECONF-FULFILL; (B) RECONF-UNFULFILL; and (C) EXCEPTION.

[28]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 [Harel, 1988]. Besides concurrency, this formalism models clustering, hierarchy and history with an excessively complex semantics, none of this required in our model.

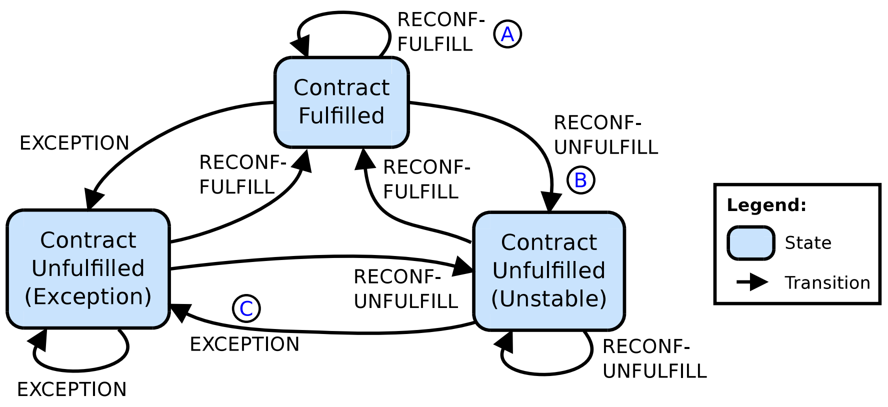


Figure 4.19: State machine for the QoS contract-preserving reconfiguration system. By design, the system starts in a *contract-fulfilled* state. Changes in the actual context condition trigger transitions (i.e., software reconfigurations).

Considering the previous observations and this figure, we give the definition of our QoSC_FSM transition function for each of the three generalized target states, *Fulfilled*, *Unstable*, and *Exception*, and corresponding generalized transition conditions, as follows.

Definition 4.9 (*QoSC_FSM* Transition Function)**.** *The transition function $\delta : CBSAR \times \Sigma \times \Delta^U \rightarrow STATES$ (cf. Def. 4.8) defines output states depending on the actual managed application reflection state ($\Re_S$), a context event (e) received from context monitors, and the execution context (Δ):*

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

For the target *Fulfilled** state, $s \,|\, e \in \eta(s)$ is the actual QoS-level state of fulfillment, while $\Psi(s \,|\, e \in \eta(s))$ 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.

4.3.5 Managing Multiple QoS Properties

In the previous sections we explained how our formal model supports the reconfiguration process for preserving different QoS levels of a same QoS property. However, our QoS contract definition allows the user to specify several QoS properties in the same QoS contract instance, and our semantics also copes well with these multiple properties. Thanks to the fact that QoS properties are orthogonal in software applications, specific design patterns can be applied independently on these applications to determine their QoS properties. In this section we illustrate how our model preserves multiple QoS properties specified in a same QoS contract. For this, we take advantage of the well separated specification of QoS properties established by our QoS contract definition, in terms of the corresponding disjoint QoS levels, triggering context events and reconfiguration rule-sets. Nonetheless, according to Algorithm 4.2, our *QoSC_FSM* semantics produces an FSM for each QoS property specification in the contract. Thus, at runtime, we need to track the current state on each of its different constituting FSMs. We characterize the execution state of the *QoSC_FSM* as a binary relation (i.e., a vector) between QoS properties and respective QoS levels.

Definition 4.10 (*QoSC_FSM* Execution State –QES)**.** *Given a QoS contract QoSC, the QoSC_FSM execution state is the tuple* $\langle CURSTATE, \Re_S \rangle$*, 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; and*
- $\Re_S$ *is the reflection structure of the managed application (CBSAR).*

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. For the videoconference system of our application scenario, the contract specifies two properties with their respective QoS levels: confidentiality and availability (cf. Table 2.1, p. 40). Thus, when initially executed from an intranet-serviced area, the *QoSC_FSM* execution state is $CURSTATE = \{(Confidentiality, clearChannel), (Availability, voiceVideo)\}$. That is, the managed software application is initially fulfilling (i) the QoS level $clearChannel$; and (ii) the QoS level $voiceVideo$.

Concerning the transitions, given that context events are notified individually by context monitors on specific context variables, a specific context event has direct effect on one specific QoS property. Hence, each *QoSC_FSM* transition is caused by the transition of exactly one of its QoS properties. Additionally, in light of multiple QoS properties, we observe that $\Re_S$ must be computed at runtime on every state change. This observation is based on the fact that the managed application can evolve dynamically from state to state on different properties, thus its structure is not necessarily the same for a same execution state. Even though we could compute statically all possible combinations of states, this would constitute a different semantics, unnecessarily more complex and, besides, unpractical. First, this semantics would have to consider the combination of each QoS level of a QoS property with all other QoS levels of all other properties. Each of these combinations should be associated with the corresponding managed application structure, resulting in a combinatorial explosion of combined states whose transitions and triggering context events would be very intricate to specify and maintain. Second, this contract semantics would have to resolve how to manage states of contract unfulfillment. Third, even if it could include the states of contract unfulfillment, this would further complicate the state transitions specification. Finally and most importantly, in contrast to our semantics, this alternative semantics would not allow dynamic re-negotiation of contracts and QoS levels at runtime (cf. Section 5.7.2, p. 102).

Algorithm 4.3 illustrates the *QoSC_FSM* execution control block, which is executed in response to reconfiguration context events.

Line 1 reconfigures $\Re_S$ and performs the state transition using the δ function. Line 2 updates the *QoSC_FSM* execution vector state (i.e., *CURSTATE*) on the property affected by e (i.e., $\pi(s \,|\, e \in qfsm.\eta(s))$), 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 $CURSTATE$.

For our example, the values for π are:

Algorithm 4.3 QoSC_FSM transition executor

```
Input: $e : \Sigma^U, qes : QES, qfsm : QoSC\_FSM$ /* QES from Def. 4.10, QoSC_FSM from Def. 4.8 */
Output: $qes$ transitioned on the property affected by $e$, using $\delta$ (Def. 4.9)
1: $S' \leftarrow \delta(qes.\Re_S, e, qfsm.\Delta)$
2: $qes.CURSTATE[\pi(s \mid e \in qfsm.\eta(s))] \leftarrow S'$
3: if $S' = Exception$ then /* a reconfiguration problem occurred (e.g., $\Re_S$ may be inconsistent) */
4:     $qes.\Re_S \leftarrow getCBSAR()$ /* recover the previous state of $\Re_S$ */
5: end if
6: return $qes$
```

- $\pi = \{(clearChannel, Confidentiality), (confidentChannel, Confidentiality), (localCache, Confidentiality), (voiceVideo, Availability), (voiceOnly, Availability), (holdCall, Availability)\}$

Hence, the context change to an extranet-serviced area makes the current execution state (i.e., $CURSTATE = \{(Confidentiality, clearChannel), (Availability, voiceVideo)\}$) to transition into $CURSTATE = \{(Confidentiality, confidentChannel), (Availability, voiceVideo)\}$.

However, given that the reconfiguration operates on $\Re_S$ and verifies its component-based structural conformance before instrumenting the respective changes in the managed application, $\Re_S$ is left inconsistent when this verification fails (i.e., the transition resulted in the *Exception* state). In this case, line 4 recovers the previous state of $\Re_S$ from the running managed application. An example of this case would result from the application of a faulty reconfiguration rule given by the user.

Finally, as the reconfiguration depends on the matching operation between reconfiguration rules and the managed application reflection structure, maintaining $\Re_S$ 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 $\Re_S$ 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.

4.3.6 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. This contract preservation is defined in terms of continuous reconfiguration cycles that start and end in actual running managed software applications, triggered by context changes. 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 4.11 (QoS Contract-Preserving Reconfiguration System –QoSCRS). *The QoS contract-preserving reconfiguration system, QoSCRS, is the tuple* $\langle \Re_S, QoSC, CBSRS, QoSC\_FSM \rangle$, *where* $\Re_S$ *is the reflection structure of the actual 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 representation QoSC_FSM and based on the context event e received, the affected QoS property (i.e., $\pi(s \,|\, e \in \eta(s))$) is identified. Then δ, the transition function is invoked, identifying the target state to be reached under the new context situation (i.e., $s \,|\, e \in \eta(s)$). Also, the corresponding QoS-level predicate and guaranteeing reconfiguration rule-set for this state are retrieved (i.e., $\Psi(s \,|\, e \in \eta(s))$ and $\rho(s \,|\, e \in \eta(s))$, respectively). With this, a state transition in the managed application reflection structure is induced, thus performing a reconfiguration $\Re_S \overset{*}{\Rightarrow} \Re'_S$ using the component-based reconfiguration system (i.e., $reconfig(\Re_S, \rho(s \,|\, e \in \eta(s)))$). From this, a reconfiguration plan is synthesized (cf. Def. 4.7).*
3. *(Pre-update checks) Once obtained $\Re'_S$, the component-based structural conformance check is performed on it. We specify the corresponding conditions in Sect. 6.6. 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 $\Re'_S$ satisfies the pre-update checks, the synthesized reconfiguration plan is applied to the running managed application, supported by f_S^{-1} (cf. Def. 4.4). 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 6.7.*

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 may select the source states of the transitions by encoding the patterns associated to these states in the LHS of the reconfiguration rules. In particular, to have a symmetric transition between a pair of states S and T, one of the reconfiguration rules used to reach S from T would have to be the inverse of the one used to reach T from S.

In this way, our QoS Contract-Preserving Reconfiguration System copes equally well with contracts having context conditions with and without inter-dependencies.

4.4 Chapter Summary

In this chapter we have presented our formal model for preserving QoS contracts at runtime through reliable and autonomous reconfiguration. We built this model by giving formal definitions for component-based software reflection, QoS contracts, and reconfiguration rules as typed attributed graphs. With these definitions as building blocks, we define our component-based structure reconfiguration system (CBSRS) as an extension of graph transformation systems. To achieve autonomous reconfiguration, our model is inspired by the MAPE-loop reference model.

Even though we conceive our model as a formal foundation for the planner component of the MAPE loop, it also involves functionalities of the monitor, analyzer and executor components. In this respect, the QoS contract is our most important definition, as it specifies not only the expected responsibilities for each of the MAPE-loop components, but also the information elements that they require to accomplish the assigned responsibilities. In light of this, we define the semantics of our QoS contracts by extending the definition of finite state machines –*QoSC_FSMs*, in which states represent expected QoS levels and transitions are performed based on software reconfiguration operations. For this semantics, we generalized the conditions for transitions to contract states of both, fulfillment and unfulfillment, thus being robust with respect to context unpredictability.

As a result, our claim is that we use (formal) models at runtime to reliably reconfigure software applications for preserving its QoS contracts. More precisely, with our model we demonstrate the feasibility of exploiting design patterns at runtime in reconfiguration loops to fulfill expected QoS levels associated to specific context conditions, while benefiting from graph transformation properties in the reconfiguration process. For this, we encode design patterns in left and right-hand sides of reconfiguration rules, given their determining influence on QoS properties. Moreover, the generalized conditions for the transitions used in the *QoSC_FSM* transition function, together with the definition of the *QoSC_FSM* Execution State, allows our model to manage multiple QoS properties defined in a same contract. For this, our model maintains the association of the current contractual state with the running software application through its reflection structure. This association is maintained in such a way that even in exception states the delivery of the managed application services is not interrupted, although in a contract unfulfillment state. Thus, our model preserves the continuity of expected QoS levels as far as the user provides adequate and relevant QoS contract specifications, which include triggering context events and reconfiguration rules. However, if the contract specification does not satisfy these conditions, our model does not interfere with the managed application execution, as it was developed. Nonetheless, in these cases our model consistently detects the causes for reaching unstable and exception states, and notifies the user about the corresponding anomalous situations.

Concerning its applicability, our formal model can be used to develop and implement rule-based self-reconfiguring software systems in automated and reliable ways, enabling them to be responsible for their QoS contracts. With these systems, a user can define her own reconfiguration rules while freeing her of being aware 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 must be self-monitoring. To this respect, proposals addressing QoS properties usually detect and manage contract violation either at a coarse-grained, system resource 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 preserve must be measurable from system components, and the corrective actions in response to their violation are also at the component-based architecture reconfiguration.

Finally, our formal model enforces the management of a clear separation of concerns, not only between the *managed application* and the *reconfiguration mechanism*, but also between their corresponding *properties*. Besides allowing us to analyze adaptation properties independently from QoS properties, this separation of concerns concedes our model platform independence and reusability.

In the next chapters we realize this formal model as a service component-based (SCA) architecture and implementation, analyze its corresponding properties, and illustrate its applicability in two application scenarios.

Chapter 5

QOS-CARE: The Realization of Our Formal Model

Contents

In this chapter we present QOS-CARE, the realization of our formal model for dynamic reconfiguration to preserve QoS contracts. To realize this model, we map our formal definitions given in the previous chapter into the MAPE-loop model elements. From this mapping, we derive the SCA components and services that comprise the QOS-CARE software architecture. In addition, we analyze and explain the design decisions necessary to implement this architecture, and complement the illustration given previously of how QOS-CARE is applied to the videoconference system of our example scenario, thus preserving its QoS contracts. The source code and design documentation of QOS-CARE is currently maintained in the INRIA GForge project repository `https://gforge.inria.fr/projects/scesame`.

To be coherent with our formal model, our design decisions for its implementation must not only be based on it, but also take into account the principles and concerns analyzed and explored by the Software Engineering for Adaptive and Self-Managing Systems (SEAMS) research community, as analyzed in Section 2.4. In the context of this chapter, these principles and concerns are related to the separation of concerns between managed applications and adaptation mechanisms, the explicitness of feedback-loop elements in the self-adaptive system architecture, and the enforcement of adaptation properties. However, defining a sound mapping from our formal model to a software design and respective implementation that preserves the formal model properties while satisfying the aforementioned principles and concerns is not a trivial problem.

First, our model comprises formal definitions based on typed attributed graphs, graph transformation, and extended state machines for modeling software structures and controlling their reconfiguration. To be useful, but also manageable, these definitions must be precise, yet abstracted from implementation details. Thus, a straight implementation from these definitions in software units using automated (or formal) methods does not necessarily guarantee, for instance, the explicitness of the MAPE-loop elements in the system architecture and implementation. In fact, despite that the MAPE loop is followed as a reference model in the design of many self-adaptive approaches, its elements are finally not explicit in their implementations. As a result, the reusability of their adaptation mechanisms is limited, as well as the possibilities for analyzing their adaptation properties independently from the managed application ones.

Second, we abstract important low-level operational actions in our formal model, which complement and refine its characteristics and properties. These operational actions and respective data structures also require a careful mapping to the MAPE-loop elements. Examples of these actions are the notification of anomalous reconfiguration situations to the user; counting the consecutive transitions to the *unstable* state before notifying the user, by inducing a transition to the *exception* state; and queuing context events if they are notified in the course of a reconfiguration operation.

In light of this, the challenge we address in this chapter is the realization of our formal model for preserving QoS contracts as a software architecture and respective implementation, maintaining the formal model properties, and following recognized design principles for self-adaptive software systems. Moreover, this implementation must be integrable with, and executed by existing component runtime platforms, implying that, ideally, our software architecture should conform to component specifications such as the SCA.

To address this challenge, in Section 5.1 we define a mapping from the elements of our formal model to the elements of the MAPE-loop reference model. This mapping is fundamental to enforce the explicitness of the MAPE-loop model elements in our implementation, and maintain the separation of concerns between the reconfiguration mechanism and the managed application at the implementation level. Based on this mapping, in Section 5.2 we derive an SCA-compliant architecture and present an overview of it, illustrating the relationships among its main components. Then, in Sections 5.3 to 5.7 we detail each of this architecture components with their assigned formal model definitions as their functionalities. For each case, we also specify their respective provided and required services, including the operational actions that complement these functionalities. Having explained our architecture components and functionalities, in Section 5.8 we present the way we conceive this realization as an additional layer for SCA middleware stacks to dynamically reconfigure component-based applications to preserve their QoS contracts. Finally, in Section 5.9 we present relevant details if the implementation

Correspondences in this Chapter: *Addressed Challenge(s):* To realize the formal model for preserving QoS contracts as a software design and respective implementation, maintaining the properties of the formal model, and following recognized design principles for self-adaptive software systems. The implementation must be executable in component runtime platforms. *Goal(s):* G5 –Determine the practical feasibility of the formal based reconfiguration mechanism implementation to preserve QoS contracts. *General contribution(s):* GC.AIE –Formal model (GC.FM1 and GC.FM2) realized and evaluated as an SCA layer for dynamic reconfiguration. *Specific contribution(s):* AIE.1 –SCA layer architecture for dynamic reconfiguration to preserve QoS contracts designed and implemented maintaining the formal model properties.

5.1 Mapping Our Formal Model to the MAPE-K Loop Reference Model

In previous chapters we (i) explained the MAPE-loop elements (i.e., Monitor, Analyzer, Planner, Executor and Knowledge manager) with their functionalities (cf. Section 2.4); based on this, (ii) identified and defined the characterizing dimensions of self-adaptive software (SAS) systems, and derived adaptation properties from this characterization (cf. Chapter 3); and (iii) presented the formal foundations for the Planner element, including particularly the QoS contract definition with its respective semantics, the central element of our reconfiguration system (cf. Chapter 4). However, even though our model is a formal foundation mainly for the Planner, the importance of this QoS contract definition –and its semantics– is that it entails responsibilities for all of the MAPE-loop elements. We identify these responsibilities in the QoS contract semantics, that is, the *QoSC_FSM* definition.

Therefore, in Fig. 5.1 we present our interpretation of how the MAPE-loop model makes explicit the monitoring, analysis, planning, execution, and knowledge management functions that are implicit in the feedback-loop block diagram.

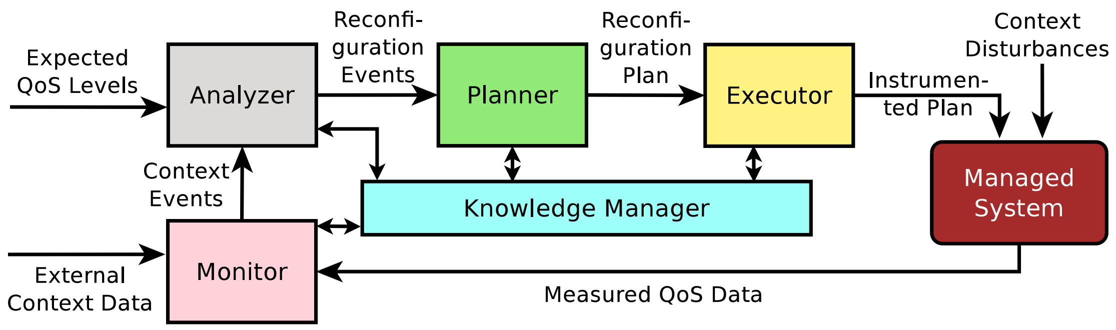


Figure 5.1: The MAPE-K loop block diagram adapted for the QOS-CARE reconfiguration loop.

Our interpretation of the feedback-loop elements and their functionalities is as follows.

Monitor. Monitoring elements are responsible for sensing variables corresponding to QoS properties from both, the managed application (i.e., *measured QoS data* in the figure), and also its *external context* (i.e., not measured from the managed application). Monitors notify *context events* to the analyzer, based on these measured values.

Analyzer. The analyzer, based on the *expected QoS-level* to fulfill according to the context events notified by monitors, determines whether a reconfiguration must be triggered. For this, the analyzer must verify the conditions to trigger a reconfiguration, such as the relevance of the events reported from the monitors, as specified by the triggering context event types declared in the QoS contract.

Planner. Once notified with a reconfiguration event from the context analyzer, the planner performs a reconfiguration on the e-graph representation of the managed application using the reconfiguration rule set associated to the QoS level to fulfill. The reconfiguration rules encode design patterns that address different QoS levels for a given QoS property and corresponding to specific context events. From this reconfiguration operation the planner synthesizes a reconfiguration plan as a list of primitive reconfiguration instructions.

Executor. Upon reception of a reconfiguration plan, the executor instruments it in the component runtime platform. This implies to translate or adequate the reconfiguration instructions to the characteristics of the particular component runtime platform in use. In our case, as QoS-CARE currently uses FraSCAti as the component runtime platform, we adequate the list of reconfiguration instructions to its reconfiguration primitives execution engine, FScript.

(Reconfiguration) Knowledge manager. A reconfiguration knowledge manager makes the relevant knowledge about the managed software application configuration and how to reconfigure it at runtime explicit, making this information available to the other MAPE-loop elements. In QoS-CARE, this knowledge is given by the user in QoS contracts in terms of context events, QoS levels to fulfill, and respective reconfiguration rules. As explained in the previous chapter, this information is encoded in the *QoSC_FSM* definition (cf. Def. 4.8, p. 81).

Following this interpretation, in Table 5.1 we present the mapping from the *QoSC_FSM* basic operations to the elements of the MAPE-loop reference model. That is, this mapping distributes the *QoSC_FSM* responsibilities in the elements of the QoS-CARE reconfiguration loop.

Table 5.1: *QoSC_FSM* basic operations distributed in the QoS-CARE reconfiguration loop

QoS-CARE Reconfiguration Loop Element	*QoSC_FSM* Formal Model Operations
Monitor	$-evalInContext : PREDICATE \times \Delta^U \rightarrow \{F, U\}$
Analyzer	– (uses) $match : CBSRR \times CBSAR \rightarrow Boolean$ – (uses) $evalInContext : PREDICATE \times \Delta^U \rightarrow \{F, U\}$
Planner	$-reconfig : CBSAR \times \mathcal{P}(\Gamma) \rightarrow Boolean$ $-match : CBSRR \times CBSAR \rightarrow Boolean$
Executor	$-getCBSAR : \rightarrow CBSAR$ $-execReconfPlan : INSTRUCTION^* \rightarrow Boolean$
Reconfiguration Knowledge Manager	$-\Psi : STATES \rightarrow PREDICATE$ $-\kappa : STATES \rightarrow PREDICATE$ $-\eta : STATES \rightarrow \mathcal{P}(\Sigma)$ $-\rho : STATES \rightarrow \mathcal{P}(\Gamma)$ $-\pi : STATES \rightarrow QoSProperty$

5.2 QoS-CARE Architecture Overview

From the previous mapping, in this section we identify the SCA components required to provide the assigned responsibilities from the formal model. We structure these components as an SCA-compliant architecture, and present an overview of them illustrating their interrelationships. We group these SCA components by each of the MAPE-loop elements as follows.

- Monitor
  - Context Monitor
  - Context Events Simulator
- Analyzer
  - Event Analyzer
- Planner
  - Reconfiguration Planner
  - E-Graph Reconfiguration Engine
- Executor
  - SCA Instrumentation
- (Reconfiguration) Knowledge Manager
  - QoS Contract Manager
  - *QoSC_FSM* Manager and Executor

Figure 5.2 illustrates the components of the QoS-CARE's SCA architecture with their exposed properties, and provided and required services. These components are shown grouped by the MAPE-loop elements (labeled M, A, P, E and K, respectively) and consider the assigned operations from our formal model as their services.

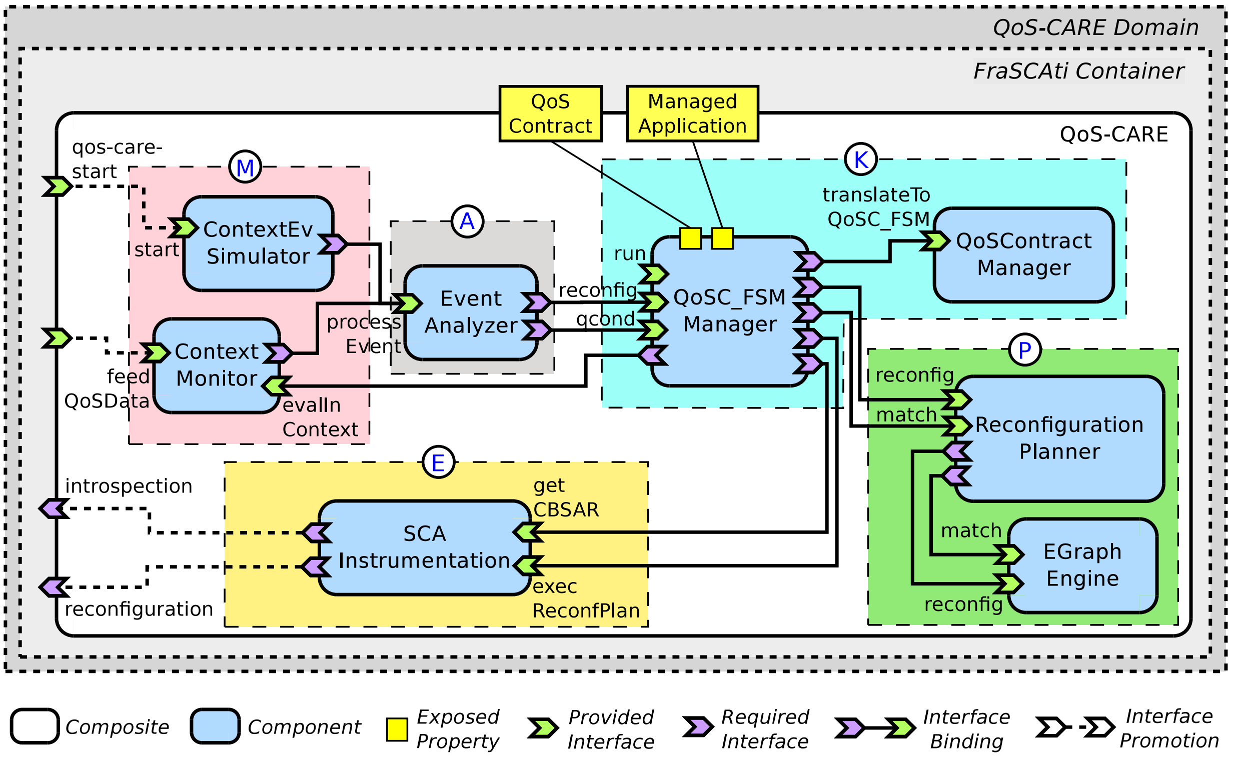


Figure 5.2: The QoS-CARE SCA-compliant architecture derived from our adapted MAPE-K model.

This architecture, which is also an SCA executable specification, exposes two properties: a QoS contract, and the component-based software application that must satisfy this contract. To configure the whole system execution's initial state, the component runtime platform (e.g., FRASCATI) loads and executes QoS-CARE. The *QoSC_FSMManager* component of QoS-CARE instructs FRASCATI to load the specified managed application; then, it loads the QoS contract and generates the *QoSC_FSM* state machine from this contract using the *translateToQoSC_FSM* service from the *QoSContractManager*. Afterwards, using the *getCBSAR* service from *SCAInstrumentation*, it obtains the SCA representation of the managed application from the FRASCATI runtime component container, and produces the corresponding e-graph component-based structure application reflection (i.e., *CBSAR*). With this structure and the initial QoS levels specified for each QoS property in the contract, the *QoSC_FSMManager* component initializes the *QoSC_FSM Execution State* (cf. QES, Def. 4.10, p. 89).

Once configured the initial QoS-CARE execution state, the managed application execution is started. Concurrently, the *QoSC_FSMManager* has the main responsibility for controlling the operations to preserve the satisfaction of the specified QoS contract, following the previously described MAPE-loop data and control flow. Nonetheless, it is worth noting that in the feedback-loop and the MAPE-loop models, the reconfiguration flows directly from the analyzer to the planner, and from this to the executor elements. In our architecture, the *QoSC_FSMManager*, a component derived from the MAPE-loop knowledge manager element, intervenes in these two points. In the first, to provide the process with the appropriate information in order to perform the expected reconfiguration according to the given context situation. This information, specified in the contract, corresponds to the affected QoS property, the target state to transition into, the QoS level to fulfill in it, and the reconfiguration rules to apply. In the second, to verify the SCA conformance of the reconfigured e-graph (i.e., the CBSAR) and update the *QoSC_FSM* execution state (QES).

Finally, being SCA-compliant, the QoS-CARE architecture components can be replaced with functionally equivalent ones with no considerable effort. In the following sections we describe each of the QoS-CARE architecture components, their services, and the interactions among them, grouped by the MAPE-loop elements. Nonetheless, it is worth emphasizing that, even though the goal of this dissertation is the preservation of QoS contracts, our strategy to achieve it is through dynamic reconfiguration. Therefore, this focuses our research work on the reconfiguration operation itself, that is, the Planner element of the MAPE loop. For the other elements we basically describe their required responsibilities, following our QoS contract definition.

5.3 Monitor

5.3.1 Context Monitor

In our contract definition (cf. Def. 4.5, p. 72), expected QoS levels and corresponding context conditions for different QoS properties are specified as predicates, whereas context event types as sets. These predicates involve variables that measure specific characteristics of the context conditions that determine the specified QoS levels. Context monitors gather these QoS measurements either directly or from *monitor probes* (also called context sensors) from the execution context and the managed application. Whenever the changes in the context modify a given context condition, the respective context monitor, specified in the QoS contract through the attribute *QoSMonitor*, must notify the corresponding context event to the analyzer.

A second requirement for context monitors is that they must implement the $evalInContext : PREDICATE \times \Delta^U \rightarrow \{F, U\}$ function, as specified in the previous chapter. This function must be the regular way of verifying whether a given condition (predicate) holds in the current (internal or external) context of the managed application.

Thus, context monitoring elements, as specified in our QoS contract and formal model definition, refer to components in the managed application that must (i) notify corresponding context events, as described previously; and (ii) implement the *evalInContext* function. These requirements are expressed in the *ContextMonitorIfc* interface, which must be implemented by context monitors to be used in QOS-CARE (cf. *ContextMonitor* in Fig. 5.2). Moreover, it is worth noting that it would be easy to modify the *evalInContext* interface and implementation to count consecutive invocations resulting in an unfulfilled condition. If this count reaches a limit, the transition function could perform a transition to a special state inhibiting further CPU-consuming reconfiguration processes that use the same ineffective reconfiguration rules.

Alternatively, if the component runtime platform implements the specified SCA capabilities for intercepting service invocations, QOS-CARE could benefit from this characteristic. However, despite FRASCATI implements this characteristic using Aspect Oriented Programming (AOP) as described by Seinturier in [Seinturier *et al.*, 2009], the automatic generation of monitor probes would require a more detailed specification, in addition to our reference to the context monitor component in the managed application. Automated monitor probe generation using AOP is a problem already solved, as reported in the PhD dissertation of González [González, 2011], whereas the automatic deployment of monitoring elements in dynamic monitoring infrastructures has been addressed in the research work of Villegas *et al.* [Villegas *et al.*, 2011a].

5.3.2 Context Events Simulator

Given that QOS-CARE focuses on the Planner element of the MAPE loop, we designed this component (cf. *ContextEvSimulator* in Fig. 5.2) with two objectives. First, to act in replacement of the components that actually should monitor contexts events in the execution environment, for testing or simulation purposes. Second, to perform "expected reliability" simulations in the spirit of the stimulus-response model, once QOS-CARE is configured with a managed application and QoS contract. These simulations would allow a system evolution architect to

i. verify the adequacy and efficacy of reconfiguration rules with respect to their SCA structural conformance and contracted QoS levels under given (simulated) context conditions;
ii. have average measurements on reconfiguration settling-times (i.e., mean-time to reconfigure, or MTTR);
iii. adjust and complete negotiable QoS contracts (i.e., sets of context events and event types to be monitored and analyzed, corresponding context conditions, QoS-level objectives and reconfiguration rules); and
iv. analyze the response under characterized context situations of given managed applications, for instance by using an event generator with a given probability distribution function.

This component also implements the *ContextMonitorIfc* interface. Thus, it can be replaced easily with actual context monitor components for final deployments. The *evalInContext* function must be implemented according to the simulation conditions and objectives.

5.4 Analyzer

5.4.1 Event Analyzer

In QOS-CARE, the event analyzer (cf. *EventAnalyzer* in Fig. 5.2) is responsible for determining whether a reconfiguration must be triggered, based on events notified by context monitors. Even though QOS-CARE allows the user to specify complex events to be notified by context monitors and, accordingly, complex reconfiguration rules to address the corresponding complex context situations, our formal model and design decisions encourage the use of single events.

First, the most basic and effective form of adaptation-triggering event is produced by single-variable context monitors measuring and evaluating one characteristic as a single value at a time. In effect, if multiple sensed values are processed (e.g., summed, correlated and operated in other ways) by higher-level context monitors and event analyzers to generate aggregated events, their impact is also on multiple contracted QoS properties. As a result, these aggregated events as such are typically useless because tracing their cause-effect relationships with the affected QoS properties and context situations is very difficult, as reported in [Yang *et al.*, 2009]. Besides, the respective reconfiguration rules are also of high complexity, being error-prone to code and maintain [Luckham, 2001].

Second, proper event processing must provide the ability of disaggregating aggregated events, tracing the relationships of causality between the aggregated event and its low-level single events. This allows to have more simple and manageable strategies that directly and specifically address the causes for single events that affect particular QoS properties.

However, we assume that these complex event processing functionalities, used to correlate and associate transient and multiple unstable events into "definitive" events, can be integrated into our event analyzer. This addresses problems such as the oscillation or unstable changes between different context situations in very short periods of time. Nonetheless, this problem is addressed in research works relating complex event processing and self-adaptation, such as [Hermosillo *et al.*, 2010], and is beyond the scope of this dissertation.

Based on the previous analysis and assumptions, and given that QoS properties of managed software applications are orthogonal among them, our event analyzer implementation uses a consumer/producer event queue to process one event at a time. Reconfiguration events notified from context monitors (i.e., the producers) are fed into this queue, while the event analyzer sends them to the *QoSC_FSMManager* component (i.e., the consumer) to processes them when no reconfiguration processes are in course. However, in this setting it is possible for an event in the queue to become obsolete, that is, its corresponding context condition is no longer valid in the moment of its processing. For this, the event analyzer uses the *evalInContext* function on the context condition (i.e., using the *QoSC_FSM*'s κ auxiliary function) that corresponds to the event in question to verify if such event is still relevant to trigger a reconfiguration. In this way, no reconfiguration events are ignored, while preventing the interruption of any reconfiguration process in course. This also contributes to manage undesirable oscillating events.

Finally, to determine if a reconfiguration must be triggered, this component uses the *match* function, factorizing it out from the *QoSC_FSM* transition function, and checks if the notified events are among the specified by the user (i.e., $e \in \Sigma$).

5.5 Planner

5.5.1 Reconfiguration Planner

This component (cf. *ReconfigurationPlanner* in Fig. 5.2) is the central element of QOS-CARE, as it implements the fundamental definitions of our formal model's reconfiguration system, in particular the $reconfig : CBSAR \times \mathcal{P}(\Gamma) \rightarrow Boolean$ and $match : CBSRR \times CBSAR \rightarrow Boolean$ operations. Thus, we conceive it as a rule-based graph-transformation system tailored for the domain of QoS contract preservation.

Receiving the e-graph representation of the managed application (i.e., a CBSAR) and the reconfiguration rule-set for guaranteeing a given QoS level, the *reconfig* operation (i) performs the reconfiguration in the managed application's e-graph representation; and (ii) synthesizes the plan to reconfigure the actual running target application as a list of primitive reconfiguration instructions, from the performed e-graph transformation operations. The instructions used in the reconfiguration plan are mappable to any set of SCA primitive operations for adding and removing components and interface wiring and bindings. To ensure the continued and consistent operation of the implied services, SCA primitives for stopping (re-starting) the affected components are included in the reconfiguration plan.

From the managed application point of view, the reconfiguration rule's left and right hand sides partially express, respectively, the current and next configuration states of their components, interfaces and bindings. Thus, the graph-based pattern matching of the rules against the managed application is the primary operation that enables the synthesis of basic blocks of instructions from which the whole reconfiguration plan is produced. This pattern matching is also the base for the *match* function.

Despite the *ReconfigurationPlanner* component is responsible for maintaining the defined structures in the QoS-CARE e-graph definition, it is designed to use existing e-graph reconfiguration engines to perform the *reconfig* and *match* operations.

5.5.2 E-Graph Reconfiguration Engine

The *E-Graph Reconfiguration Engine* component (cf. *EGraphEngine* in Fig. 5.2) is the responsible for providing the actual services implementing the *reconfig* and *match* operations, either by itself, or embodying an existing e-graph reconfiguration engine. In the latter case, this component adapts the corresponding QoS-CARE structures to the defined by the reconfiguration engine used, and vice versa.

In our current implementation, for the e-graph pattern matching, transformation and synthesis of the reconfiguration plan, we are using a modified version of the Attributed Graph Grammar open source system (AGG, [Taentzer, 2004]).

5.6 Executor

5.6.1 SCA Instrumentation

Instrumentation makes reference to two functions: (i) the gathering of information from the managed application; and (ii) the possibility to modify it. In QoS-CARE, the SCA instrumentation component (cf. *SCAInstrumentation* in Fig. 5.2) provides the operations $getCBSAR : \rightarrow CBSAR$ and $execReconfPlan : INSTRUCTION^{*} \rightarrow Boolean$ to accomplish these functions. Thus, specific implementations of this component allows QoS-CARE to interact with any SCA-compliant runtime platform with basic reflection and reconfiguration capabilities.

The *getCBSAR* operation provides the functionality of retrieving the SCA structure of the managed application, as maintained by the component runtime platform executing it. *execReconfPlan* receives a list of instructions as a reconfiguration plan, and translates them to the particular reconfiguration primitives specified by the component runtime platform in use. In the case of the implementation of this component for the FraSCAti platform, this operation translates the reconfiguration instructions to FScript, the FraSCAti's reconfiguration primitives execution engine.

For the design of this component we considered the self-adaptation necessities in ubiquitous, distributed and mobile devices and applications, as well as the computing-power requirements of component platforms and their reconfiguration operations. In order to allow distributed reconfiguration and balance the computational power requirements, we modified the reflection interfaces in FraSCAti to be accessible as remote services, as illustrated in Fig. 5.3. In this way, FraSCAti can be deployed in each required machine with *RemoteIntrospection* services, accessible both locally and remotely.

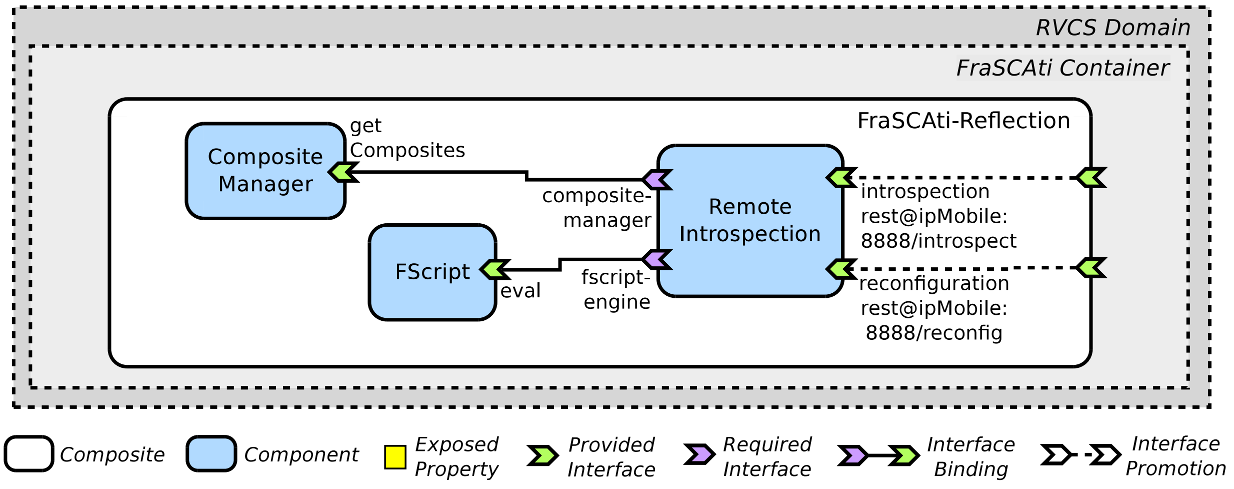


Figure 5.3: The FRASCATI reflection subsystem with remote invocation services.

5.7 Knowledge Manager

5.7.1 QoS Contract Manager

The QoS contract manager (cf. *QoSContractManager* in Fig. 5.2) is responsible for the following functionalities:

i. load the QoS contract to be satisfied;
ii. translate it to the *QoSC_FSM* structure using Algorithm 4.2; and
iii. initialize the *QoSC_FSM* execution state.

Having these functionalities in this component allows QOS-CARE to have different and independent representations (i.e., file formats) for QoS contracts. Currently, we support XML and AGG file definitions.

Given that all of the QoS contract attributes are translated to the *QoSC_FSM*, this component remains unused until the contract is renegotiated, as explained in the following section.

5.7.2 *QoSC_FSM* Manager and Executor

The *QoSC_FSM* manager and executor component (cf. *QoSC_FSMManager* in Fig. 5.2) is the main QOS-CARE's orchestrating element. It has all the relevant knowledge to answer what, how, and when to reconfigure a given managed software application. This knowledge is obtained from the QoS contract specification, which is translated into the *QoSC_FSM* definition, using the *QoSContractManager* component. Then, this knowledge is made available by the *QoSC_FSMManager* to the other MAPE-loop elements through corresponding services (not shown in the figure for readability reasons).

The *QoSC_FSMManager* component provides the following services:

- $reconfig : \Sigma^U \rightarrow Boolean$ implements Algorithm 4.3 (i.e., the *QoSC_FSM* transition executor) and applies it upon a reconfiguration request caused by a context event notification.
- The *QoSC_FSM* auxiliary functions, to obtain respectively:
  - $\Psi : STATES \rightarrow PREDICATE$: the QoS level to fulfill on the target state;
  - $\kappa : STATES \rightarrow PREDICATE$: the corresponding context condition;

  - $\eta : STATES \rightarrow \mathcal{P}(\Sigma)$: the type of the context events that trigger transitions to the target state;
  - $\rho : STATES \rightarrow \mathcal{P}(\Gamma)$: the guaranteeing reconfiguration rule-set to fulfill the QoS-level objective;
  - $\pi : STATES \rightarrow QoSProperty$: the affected QoS property.
- The *QoSC_FSM* transition function $\delta : CBSAR \times \Sigma \times \Delta^U \rightarrow STATES$, which uses the previous functions and actual managed application's e-graph reflection structure (i.e., $\Re_S$) to request the *ReconfigurationPlanner* to perform a reconfiguration. After successfully performing the *pre-update checks*, it invokes the *execReconfPlan* service of the *SCAInstrumentation* component to instrument the reconfiguration plan in the managed application.

In summary, this component uses the QoS contract information as knowledge to determine the reconfiguration rules to be applied in the managed application in order to satisfy the QoS-level objective to fulfill when context conditions change.

5.8 QoS-CARE as an SCA Layer for Preserving QoS Contracts

Having presented the architecture of QoS-CARE, in this section we illustrate how does it fit as an additional layer in the FraSCAti's middleware stack to preserve QoS contracts in executed applications.

Middleware is defined as the software (macro)layer that lies between software applications and the operating system across a distributed computing system [Krakowiak, 2009]. As a form of operating system extension, its purpose is to provide common programming abstractions as generic functions to software applications, such as transparent remote method invocations (i.e., independent of network protocols), logging of operations, and data translation for service interoperability (i.e., independent of syntax and format).

In Fig. 5.4 we recall the FraSCAti's four layer middleware stack, introduced in Section 2.2.3. In this stack, each layer adds specific functionalities or capabilities to the ones below it. The personality layer adds service interception, and life-cycle management capabilities to the components defined in the kernel layer. The run-time layer, responsible for managing the components container, provides functionalities for loading and checking application composites. It also adds introspection and primitive reconfiguration capabilities to query about the components and their interfaces and services, and to add and remove them at run-time. Finally, the non-functional layer adds basic support for non-functional requirements to the executed applications.

As illustrated in the figure, we add QoS-CARE as a fifth layer of FraSCAti, an SCA middleware implementation, to preserve QoS contracts in its executed applications. In effect, driven by a contract specification, and exploiting the functionalities of the four layers below it, QoS-CARE provides autonomous and reliable reconfiguration services to maintain fulfilled the expected QoS levels under the respective contexts conditions.

5.9 Implementation Details

In this section we provide some implementation details underlying the QoS-CARE architecture components in terms of its classes, physical lines of code (LOC), and density (i.e., LOCs per class). Although we wrote most of the code in Java, we also used the C programming language for some functionalities. Nonetheless, since the portions of C code are small compared to the

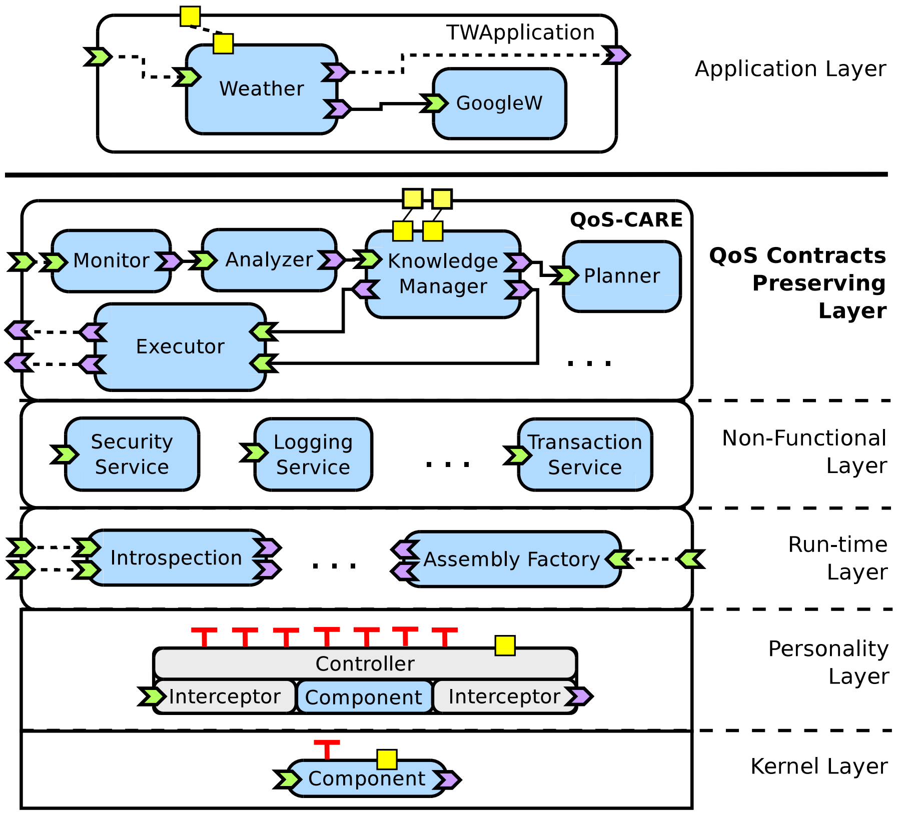


Figure 5.4: QoS-CARE as a layer for preserving QoS contracts in the FraSCAti middleware stack.

rest of the implementation in Java, we sum their lines of code without loss of accuracy. Table 5.2 presents a summary of these implementation details.

Table 5.2: QoS-CARE Implementation Details

QoS-CARE Reconfiguration Loop Element	Components	Classes and Interfaces	LOC	LOC/ Class
Common Core	0	25[a]	2,618[a]	104
Monitor	2	6	1,287	214
Analyzer	1	2	380	190
Planner	2	4	1,721	430
Executor	2	3	426	142
Reconfiguration Knowledge Manager	2	14	2,846	203
Total	9	54[a]	9,278[a]	171

[a] Includes modules and files in C, Yacc and Lex.

As illustrated in this table, we group these details by the elements of the MAPE-loop model adapted for the QoS-CARE reconfiguration loop (cf. Section 5.1), as follows.

Common Core. This package comprises the QoS-CARE's core data structures and respective operations that we use to integrate the functionalities of its other components. The common core has:

- 18 Java classes for implementing:
  - The QoS-CARE definition for the SCA component assembly specification: we use this definition as a pivot to translate the particular SCA platform implementation representation (e.g., the implemented in FraSCAti) of the managed application to our e-graph representation. This definition is necessary to maintain the QoS-CARE independence of different SCA implementations.
  - Context management basic definitions: provide general event types for the events to be notified by context monitors to context analyzers.
  - Utilities: provides functionalities for the generation and loading of reconfiguration plans, and logging and execution of other auxiliary operations.
- 7 modules in C, and file specifications in Yacc and Flex compiler/interpreter construction tools to generate the runtime evaluator of predicates (i.e., QoS levels and context conditions).

Monitor. We group in this package the basic definition of monitoring elements responsible for sensing QoS properties (one Java interface), and the context-events simulator (5 Java classes). The interface for monitors declares the *evalInContext*, *sense* and *notifyContextEvent* method signatures. The context-events simulator implements:

- A function for generating context events. Although it is possible to implement functions with different probability distributions, to evaluate the settling time (cf. Chapter 7) we implemented a uniform distribution function to stress the system with a uniform load.
- A queue for storing and processing the generated context events.

Analyzer. This package uses the functionalities of the *QoSC_FSMManager* to retrieve the contract information and determine whether a context event must be added to the queue of reconfiguration events. Based on the status of this queue and the current context conditions, the event analyzer notifies the *QoSC_FSMManager* with the next reconfiguration event. As illustrated in Table 5.2, this package is the smallest one, with 380 lines of code distributed in one Java interface and one Java class implementing one component.

Planner. This package includes one Java interface and 3 Java classes to implement the following functionalities:

- In the reconfiguration planner: the bridging operations for:
  - Loading the typing structures for component-based software applications and QoS contract specifications (including the e-graph based reconfiguration rules), and their corresponding instances.
  - Invoking the *match* and *reconfig* operations, translating the respective parameters from the QoS-CARE definitions to the ones of the e-graph specific implementation used.
  - Generating the reconfiguration plan from the e-graph transformation operations, using the code generation data structure from the QoS-CARE common core.
- In the e-graph engine: this component encapsulates the e-graph (i.e., typed attributed graph) implementation. In the current implementation of QoS-CARE, we use a modified version of the Attributed Graph Grammar (AGG) system[29], which implements the required graph definitions, including the *match* and *reconfig* operations as specified in the previous chapter.

[29]http://user.cs.tu-berlin.de/~gragra/agg/

Executor. This package implements the executor functionalities with one Java interface and two Java classes. The interface defines the SCA instrumentation methods for (i) obtaining the SCA representation of the managed application, and (ii) executing the reconfiguration plan. In the current implementation of QoS-CARE we provide two SCA instrumentation components with the same implementation, the first for local access, and the second for remote instrumentation access.

(Reconfiguration) Knowledge manager. This package contains 3 Java interfaces and 11 Java classes, which implement the data structures and functionalities for:

- The *QoSC_FSM* structure, into which the whole QoS contract is translated from its e-graph representation, using Algorithm 4.2. This includes the reconfiguration rules and the mappings for the *QoSC_FSM* auxiliary functions.
- The execution vector state of *QoSC_FSM*, and its transition executor function, to control the managed application state with respect to the contracted conditions.

5.10 Chapter Summary

In this chapter we have presented the architecture of QoS-CARE, the most significant decisions we made for its design, and some details of its implementation.

Concerning the contributions of this chapter, we summarize them as follows. We showed the implementation feasibility of our formal model by developing it as an SCA software design and respective implementation. This implementation not only follows our formal model definitions, but also satisfies important principles established by the software engineering for self-adaptive systems community. Instead of directly deriving or implementing the formal definitions into software units, we first mapped these definitions to the MAPE-loop model elements. Then, from this mapping, we synthesized in SCA components the functionalities and services required to satisfy the formal definitions. Thus, the MAPE-loop elements are explicitly identifiable as the SCA components comprising the adaptation mechanism, effectively allowing to manage a clear separation of concerns between it and the managed software application. As a result, while retaining our formal model definitions, the adaptation mechanism implementation can be independently (re)used and integrated with different component runtime platforms, and moreover, evaluated and analyzed separately in its adaptation properties.

In this setting, it is worth noting that the use of classical formal development methods for deriving certified software implementations of our formal model would be limited. As explained before, using these methods would not guarantee the aforementioned principles established by the software engineering for self-adaptive systems community. To solve this problem we would need a tailored formal method of software development to satisfy domain-specific software engineering principles and requirements (e.g. for self-adaptive systems). Naturally, this problem should be addressed by the formal development and programming methods community in conjunction with the MODELS@runtime and the SEAMS communities.

Finally, instrumenting the reconfiguration operations that result from our graph-based reconfiguration system requires capabilities of introspection and dynamic reconfiguration in the actual running software application, not easy to implement. However, the component-based development paradigm propose the more general strategy of managing these capabilities, not at the application level nor at the operating system level, but at an intermediate level, namely, the middleware level. QoS-CARE benefits from this intermediate level, which is the *component runtime platform*, and finds its place in it in the form of an added layer.

In the next chapter we analyze the reliability of our formal model and its implementation. We define this reliability in terms of five of the adaptation properties introduced in Chapter 3.

Part III
Validation

Chapter 6

Validation and Verification of QoS-CARE Properties

Contents

In the previous chapters we have presented the formal definitions that constitute our model for preserving QoS contracts, as well as its corresponding realization as a service component architecture. In this chapter we present the validation and verification of the properties that result from these formal definitions and the design decisions we made for their realization.

As in many approaches to realize self-adaptive software (SAS) systems, we assume that given a software application subject to expected QoS levels under specific context conditions, the software application must be unavoidably reconfigured whenever the context condition changes, implying the fulfillment of a different QoS level. The objective of this reconfiguration is, of course, to fulfill the QoS level that corresponds to the new context condition. However, we consider also of fundamental importance that this reconfiguration be guaranteed in its reliability in order to avoid compromising the application of the user-defined reconfiguration rules. This is important because our model guarantees the continuity of agreed services delivery as far as the user provides relevant and adequate reconfiguration rules to cope with context changes. Thus, this reliability aims at ensuring the continuity of service delivery while fulfilling the contracted QoS levels, specially under changing context conditions. This implies that the reconfiguration process must also be robust with respect to context unpredictability, managing not only states and conditions of contract fulfillment, but also of unfulfillment, as explained in the previous chapters.

Consequently, in this chapter we address the specific challenge of guaranteeing the reliability of the software reconfiguration process to preserve QoS contracts. For this, we first refine the classic definition of reliability in light of adaptation properties related to fundamental problems that self-adaptation poses to software systems, and then we show how do we validate and verify these properties. These properties are the short settling-time, the reconfiguration termination, the robustness with respect to context unpredictability, the component-based (SCA) structural conformance, and the reconfiguration atomicity. The importance of these properties is that they guarantee that reconfiguration processes (i) finish; (ii) when finished, the resulting e-graph software structure is fully component-based conformant; (iii) once successfully verified, the actual running software state is updated with the reconfiguration changes (being this an atomic process); and (iv) the final running-software state is in a controlled contract state, being this a state of either, QoS level fulfillment, if all reconfiguration steps were performed successfully, or contract unfulfillment, in any other case.

Concerning *when* and *where* can we validate and verify the aforementioned properties, and to what extent, we address the following questions:

i. What properties can be exclusively verified at design time (executing neither the managed system nor the reconfiguration mechanism)?
ii. What properties can be exclusively verified or tested at runtime?
iii. What properties can be verified or tested either at design time or at runtime?
iv. What properties can be exclusively verified or tested in the managed system?
v. What properties can be exclusively verified or tested in the reconfiguration mechanism?
vi. What properties can be verified or tested either in the managed system or in the reconfiguration mechanism?

This chapter is organized as follows. In Section 6.1 we illustrate how our formal model enforces independence and separation of concerns between reconfiguration mechanisms and managed software applications. In Section 6.2 we establish our definition of reliability in terms of adaptation properties. In Sections 6.3 to 6.7 we present the definitions for each of the addressed properties and how our model guarantees them.

Correspondences in this Chapter: *Addressed Challenge(s):* C3; C4 –The reconfiguration mechanism must be clearly separated from the managed software application, as well as their corresponding properties; Uncertainty must be managed robustly with respect to the unpredictability of context events faced by the managed application, as well as the parameterized reconfiguration rules in the reconfiguration mechanism. *Goal(s):* G3, G4, G5 –Maintain a clear separation of concerns between the reconfiguration mechanism and the managed software application, as well as between their corresponding properties; Guarantee robustness in the reconfiguration mechanism with respect to possible and foreseeable situations associated to the management of the unpredictable nature of context; Determine the practical feasibility of the formal based reconfiguration mechanism. *General contribution(s):* GC.AIE –Formal model (GC.FM1 and GC.FM2) realized and evaluated as an SCA layer for dynamic reconfiguration. *Specific contribution(s):* AIE.2 –Formal model's proof-of-concept implementation experimentally evaluated in a real SCA platform; practical feasibility and (re)usability of reconfiguration mechanism determined.

6.1 Reconfiguration Independence and Separation of Concerns

Self-adaptive software (SAS) systems are composed of two well identified parts: the adaptation (i.e, reconfiguration) mechanism and the managed software application to be adapted (i.e, reconfigured). However, as analyzed in Section 2.4, the lack of explicitness and visibility of feedback-loops in most SAS systems renders their adaptation mechanisms as non-reusable and unanalyzable in their respective properties and real advantages. One of the main causes for this problem is the intertwined exploitation of different realization techniques on both the self-adaptation controller and the managed application, thus blurring their respective limits.

In the case of our formal model to preserve QoS contracts in managed software applications, the reconfiguration strategy is built on e-graph models. The e-graph representation of running managed applications establishes precise limits between the reconfiguration mechanism and the managed application, and moreover, of the component platform that executes them. Furthermore, this representation, combined with our e-graph modeling of QoS contracts, allows us to manage a clear separation of concerns between our reconfiguration mechanism and the managed software application, and their corresponding properties. The key factor to achieve this independence and separation of concerns is the indirect relationship between *QoSProvision* attributes and reconfiguration rules (cf. Def. 4.3, 4.4 and 4.6). We use *QoSProvision* attributes to identify the managed application elements that are potential reconfiguration objectives. These elements can refer to components, interfaces and bindings in both, the managed application reflection structure, and left and right hand sides of reconfiguration rules. Thus, the relationships between the left-hand sides of these rules and managed applications 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 our reconfiguration mechanism from the managed application. In turn, this decoupling explicitly enforces the clear separation of concerns between the contractual QoS properties on the managed application and the properties of our reconfiguration mechanism.

A second factor supporting the independence and reusability of the reconfiguration mechanism based on our formal model is the central role of our QoS contract definition as our model's coherence enforcer with respect to the QoS properties of interest (cf. Def. 4.5 and 4.7). This coherence, related to context attributes, supports the collaborative and coordinated work among our model elements, but also between the model itself and the monitoring, analysis and executor MAPE-loop elements. In this way, our contract definition serves several objectives: (i) it specifies QoS obligations of a managed application to its users; (ii) it establishes the responsibilities for our planner's internal components to fulfill these obligations; and (iii) it declares the responsibilities for the monitor, analyzer and executor elements that would be required by our model to complete a MAPE-loop in a final deployment.

6.2 Reliability in the Context of Self-Reconfiguration

A first consideration to tackle when addressing self-reconfiguration reliability is the definition of reliability itself. This is explained because despite the several definitions with the different meanings that have been used for it, self-adaptation as a research area requires its own definition for this characteristic [Villegas *et al.*, 2011b].

In the context of dependable computing, [Avizienis *et al.*, 2004] presented a thorough set of characterized definitions for dependability, its related sub-properties, threats, and means for their achievement. In particular, they defined reliability as the *continuity of (agreed) expected service delivery*, and identified the (fallible) determination of possible causes of failures in service delivery as a non-obvious problem. Nonetheless, they defined *failure* as any deviation from the service delivery in the way it is agreed; and *failure modes* as the different manifestations of these deviations. In the same line, reliability has been traditionally measured in relation to the response that a system exhibits to its *own failures*, that is, in the sense of system trustworthiness [Reussner *et al.*, 2003, Candea *et al.*, 2004, Yacoub *et al.*, 2004, Filieri *et al.*, 2010, Huang *et al.*, 2011]. These definitions are in agreement with those given by the ISO/IEC 9126, the standard for the evaluation of software quality, and the Software Engineering Institute (SEI), that is, the *capability of software to maintain its level of performance under stated conditions for a stated period of time* [Barbacci *et al.*, 1995, ISO, 2001].

In contrast to these definitions, the engineering of self-adaptive software systems is directly concerned with goals such as self-healing, self-recovery and self-protection [Salehie and Tahvildari, 2009]. These goals clearly address the causes for foreseeable failures in a direct way, just as failure modes are considered when designing evaluation tests for reliability. Furthermore, in our case the continuous fulfillment of expected quality of service (QoS) levels may be interrupted or violated not only as a consequence of unexpected system failures, but also of natural changes in the context conditions of system execution. Thus, one question naturally arises: is a self-healing or self-recovery system automatically (fully) reliable? since the answer is of course negative, we further refine this question to argue that in the context of self-adaptive software, some characteristics, such as the reliability, must be reconsidered. In particular, we must consider the causes that affect the reliability in at least three cases: in the managed application, (i) for the cases directly covered by self-* goals; but also (ii) for the uncovered cases; and (iii) in the adaptation (reconfiguration) process itself. These cases should be considered not only for classifying the aforementioned failure modes, but more importantly, for considering context changes as another cause for disrupting the continuity of agreed services in their expected levels. Therefore, according to the goals pursued by this dissertation, we define and evaluate the reliability as the *continuity of expected QoS levels fulfillment in software application services when facing not only natural context changes, but also unexpected anomalous situations* (e.g., faulty reconfiguration rules given by users).

A second consideration for our definition of reliability aims at assessing it consistently. Given that our model is inspired by the MAPE loop –a reinterpretation of the feedback-loop, a consistent set of properties to evaluate its reliability would be a corresponding set of reinterpreted properties used to evaluate feedback-loops. Hence, we define the reliability of our reconfiguration system in terms of five of the adaptation properties that we characterized in Chapter 3. We selected these properties as they consistently fulfill the enunciated considerations for reliability in self-reconfiguring software systems, as defined in the following section.

6.2.1 Reliability in Terms of Adaptation Properties

We define the selected adaptation properties that characterize our concept of reliability as follows:

i. Short settling-time: the time that the reconfiguration mechanism takes for performing the self-reconfiguration must be acceptable considering the application domain. Thus, defining settling-time in absolute terms is not only unpractical, but also unfeasible for all application domains. This property, originally defined by control theory for feedback-loops, is equivalent in our case to the Mean Time to Reconfigure (MTTR), the most critical factor determining reliability, as measured quantitatively, according to the previous definitions.

ii. Reconfiguration termination: the process performed for reconfiguring the managed software application to preserve the QoS contract is guaranteed to terminate. This property is derived from the short settling-time and complements it when applied in the domain of software systems. In contrast to physical target systems controlled by computing continuous mathematical functions, the structural reconfiguration of software systems requires sequences of discrete operations to be applied on them. Thus, although not required in classical feedback-loops for physical target systems, termination is a fundamental property for the reconfiguration control of software systems.

iii. Robustness with respect to context unpredictability: the delivery of the managed software application services must remain unaltered even if the managed application state differs from the expected (contracted) state in some measured way. In this definition, the states are those determined by the context. Also, the reconfiguration process is robust if the reconfiguration mechanism is able to operate consistently (within desired limits) even under unforeseen context conditions. This implies to manage both, context events notified by context monitors but unforeseen by the user, as well as the inefficacy or non-existence of user-defined reconfiguration rules to fulfill specific QoS levels. Thus, qualitatively, robustness is the most critical factor for reliability.

iv. SCA structural conformance (structural consistency): the conformance of the managed application with respect to the structural integrity constraints defined by the component-based (SCA) specification is preserved after each reconfiguration. This property is critical to avoid failures caused by faulty reconfiguration rules defined by users, thus contributing to enforce reliability but also robustness.

v. Atomic reconfiguration: the reconfiguration is completed wholly and successfully, or it fails and the software system is left in its previous consistent state. This property is critical to enforce consistency and robustness, preventing the managed application to reach undesirable and inconsistent states.

6.2.2 Design-time vs. Run-time Validation and Verification of Properties

In the following table we summarize *when* (design-time vs. run-time) and *where* (reconfiguration mechanism vs. managed application) do we validate and verify the previously defined properties.

Table 6.1: When and where QoS-CARE properties are validated and verified

	Design-time V&V	Run-time V&V
Reconfiguration Mechanism	• Atomicity • Robustness	• Settling-time • Termination
Managed Application	(Not Applicable)	• SCA Structural Conformance

Atomicity and robustness are properties validated in the reconfiguration mechanism as a result of our formal model, being its conditions sufficient to be verified at design-time. In contrast, to be sufficient, the necessary conditions for guaranteeing the properties of termination and SCA structural conformance, shown in our formal model, require to be verified at run-time on their actual corresponding instances. That is, on the reconfiguration rules given by the user in the case of termination (i.e., in the reconfiguration mechanism); and on the actual e-graph representation of the running software system in the case of SCA structural conformance (i.e., in the managed software application). We measure the settling-time experimentally by averaging it at runtime using performance benchmarks on the reconfiguration mechanism for our application domain, as a reasonable guarantee for the expected MTTR.

Even though termination can be argued to be verified at design-time, two considerations led us to verify it at run-time. First, in contrast to the conditions for atomicity and robustness, which hold for any QoS contract, the termination conditions depend on the set of reconfiguration rules. That is, these conditions must be verified on each set of parameterized rules, which are part of QoS contracts. Second, even if verifying these conditions for a particular contract and reconfiguration rules is considered design-time, new QoS levels introduced as a result of contract re-negotiation definitively require this verification to be performed at run-time. Similar reasoning applies for the SCA structural conformance property.

In the following sections we show how our formal model and its realization guarantee the aforementioned properties for the reconfiguration mechanism, even if the property is verified in the managed application, as we analyzed in Chapter 3.

6.3 Short Settling-Time

The aforementioned definition of reliability given by Avizienis *et al.* as *continuity of (agreed) expected service* makes the short settling-time in self-reconfiguration systems as the most critical factor determining it when evaluated quantitatively. In effect, minimizing the reconfiguration settling-time implies to maximize the continuity of the managed application services with the expected (contracted) QoS levels. However, it is worth emphasizing that the definition of Avizienis *et al.*, as well as other classic definitions of reliability, focus on system failures as the main causes for interrupting or deviating the services delivery in their agreed conditions. In contrast, we focus mainly on context changes as the primary causes for these interruptions or deviations, and faulty QoS contract specifications (e.g., context events and reconfiguration rules) given by users, as secondary ones. Furthermore, we identify these deviations as sources for failures, and associate them as necessary conditions to the verification of robustness with respect to context unpredictability.

We measure the settling-time as the mean time to reconfigure (MTTR) metric by actually executing QOS-CARE to reconfigure a given managed application subject to a particular QoS contract, in order to provide realistic estimations for the user to determine its adequacy and acceptability. To obtain the measurements required to compute the MTTR we use the *Context Events Simulator* of our reconfiguration mechanism to perform "expected reliability" simulations as explained in Section 5.3.2.

In particular, in Sections 7.2.2 and 7.3.5 we analyze the MTTR measurements obtained for the two application scenarios that we use to validate the applicability and practical feasibility of QOS-CARE. This analysis shows the adequacy and acceptability of the obtained settling-times for the respective application domains (i.e., mobile client-server software applications and Internet mashup applications).

6.4 Reconfiguration Termination

Heckel *et al.* and Ehrig *et al.*, among others, showed several graph transformation theorems and results as valid also for typed attributed graph transformation systems (TAGTS) in [Heckel *et al.*, 2002], [Ehrig *et al.*, 2004], and [Ehrig *et al.*, 2009]. They carefully discuss the conditions on which these theorems and results are based, as well as their implications for the local confluence, confluence and termination properties of this kind of graph-transformation systems.

In this section, we focus on the termination conditions of our component-based structure reconfiguration system (CBSRS). We show that its most critical *CBS reconfiguration step* (i.e.,

$G_0 \overset{*}{\Rightarrow} G_n$ in *CBSRS*, Def. 4.7) is reducible to a typed attributed graph transformation system, as follows.

Theorem 6.1 (Reducibility of the CBS Reconfiguration Step)**.** *Let $CBSRS$ be a component-based structure reconfiguration system according to Def. 4.7. Every CBS reconfiguration step in $CBSRS$ is reducible to a typed attributed graph transformation system, $TAGTS$.*

Proof. According to Def. 4.7, a $CBSRS$ is a tuple $(DSig, CBS, \Re_S, P)$. Of these elements, during the *CBS reconfiguration step* (i.e., $G_0 \overset{*}{\Rightarrow} G_n$), the data signature, $DSig$, the component-based structure definition, CBS, 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, $\Re_S$, and the set of reconfiguration rules, P. Additionally, given that

1. a Component-Based Structure Application Reflection (*CBSAR*, Def. 4.4) is a tuple (G, f_S, t), where G is the e-graph that represents the software application S through the one-to-one function $f_S : S \to G$, and t an e-graph morphism $t : G \to CBS$, hence in the *CBS reconfiguration step* f_S is 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, $u : AG \to ATG$, where ATG is a type graph; and
3. a component-based reconfiguration rule, p, according to Def. 4.6, is a tuple $(L, K, R, l, r, lt, kt, rt)$, $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, with $lt : L \to CBS$, $kt : K \to CBS$ and $rt : R \to CBS$; and
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 *CBS* and the typed attributed graphs are based on the same e-graph definition,

then, every *CBS reconfiguration step* 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 component-based reconfiguration rules without the lt, kt, rt morphisms, given that, once defined the component-based reconfiguration rules, these morphisms are also fixed. $\square$

As a result, the theorems that are valid for the TAGTS are also valid for our reconfiguration system. In particular, we benefit from termination theorems and results on conditions that graph transformation rules must hold for this property. We apply these conditions specifically for our QoS-contracts preservation domain following the criteria for layered TAGTS established in [Taentzer, 2004, Ehrig *et al.*, 2005, Bruggink, 2008, Bucchiarone *et al.*, 2009, Ehrig *et al.*, 2009]. By checking these conditions we determine whether the *CBS reconfiguration step*, parameterized with the user-defined reconfiguration rule-sets in our reconfiguration system, is terminating. In other words, once parameterized with a specific set of rules, the reconfiguration system can identify termination conflicts in the rules. The absence of these conflicts determines that the process of rule application is guaranteed to finish.

Ensuring reconfiguration termination, besides contributing to the continuity of service delivery fulfillment with the expected QoS levels under varying context conditions, frees a software evolution architect of (i) being aware of rule dependencies that may cause deadlocks or infinite loops in the reconfiguration process; and (ii) coding the specific procedures for applying the reconfiguration rules and perform the reconfiguration itself.

6.5 Robustness with Respect to Context Unpredictability

Our definition of 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 prove that QoS-CARE guarantees robustness, it is sufficient to show that our model considers and controls all 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).

Our proof is as follows. In Section 4.3.4 we defined our QoS contract semantics in terms of an extended state machine, the *QoSC_FSM*. This semantics interprets the QoS levels defined in contracts as states with conditions to be fulfilled using reconfiguration rules in the corresponding transitions. Our semantics also adds two states of contract unfulfillment to this interpretation, namely, the states of exception and unstability. Finally, this semantics defines the transition conditions for these two states as the complement of the generalized conditions for the transitions to the contract fulfillment states, as expressed in the definition of the δ transition function (cf. Def 4.9).

In other words, the expected states for managed applications 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 confront in its execution, as foreseen by the user. However, given the unpredictable nature of context changes (e.g., unexpected amounts of users virtually attending a video-streamed music concert or the final match of the soccer world-cup), it is easy to presume that the user can underestimate them. Of course, this situation originates the possibility for the managed application to reach unexpected and undesirable states, caused by context changes mistakenly omitted by the user. Nonetheless, even in these states the managed application is under controlled states that are automatically generated by our model.

In this way, our model considers all possible states with their respective transitions that a managed software application can reach in its execution, with respect to context (i.e., as specified in QoS contracts). Hence, we can conclude that the robustness conditions are validated in our QoS contract semantics, as part of our reconfiguration mechanism.

6.6 Component-Based Structural Conformance

Component-based structural conformance is defined as the conformance that the managed application structure must have with respect to the structural integrity constraints given by the service component architecture (SCA) specification. This property prevents that malformed application structures, caused for instance by faulty reconfiguration rules, will cause severe execution exceptions or faults. We define this property as follows.

Definition 6.1 (Full CB-Structural Conformance)**.** *A runtime system reflection structure, $\Re_S$, is fully CB-structural conformant if it is a component-based structure (i.e., there exists a graph morphism $t : \Re_S \rightarrow CBS$), and the following structural integrity conditions hold:*

1. $\forall b1, b2\big((b1, b2 \in \Re_S.Binding \wedge b1.provided = b2.provided \wedge b1.required = b2.required) \implies b1 = b2\big)$*: every binding must connect different pairs of provided-required interfaces.*

2. $\forall b \exists i, j\big(b \in \Re_S.Binding \wedge i, j \in \Re_S.Interface \wedge i \neq j \wedge b.provided = i \wedge b.required =$

$j \wedge i.signat = j.signat)$: *each binding connects different interfaces that, nonetheless, have the same signature.*

3. $\forall i(i \in \Re_S.Interface \implies \exists c(c \in \Re_S.Component \wedge (c.ifcp = i \vee c.ifcr = i)))$: *every interface must belong to one component.*

4. $\forall i \exists c\big((i \in \Re_S.Interface \wedge c \in \Re_S.Component \wedge c.ifcr = i) \implies \exists b, j, d(b \in \Re_S.Binding \wedge b.required = i \wedge b.provided = j \wedge j \in \Re_S.Interface \wedge d \in \Re_S.Component \wedge d.ifcp = j \wedge c \neq d)\big)$: *all interfaces required by a component must be bound to interfaces provided by another component.*

The verifiability of full CB-structural compliance naturally results from Def. 4.3 and 4.4. In particular, in Sections 7.2.2 and 7.3.5 we verify that the specified conditions are satisfied by the reconfiguration rules of the two application scenarios presented in the next chapter to validate the applicability of QOS-CARE.

6.7 Atomicity of the Reconfiguration Process

Our definition for the atomicity property on self-adaptive software systems states that, either the reconfiguration process finishes successfully and the system is reconfigured, or it fails and the system preserves its (previous) configuration and state. Our proof showing that this property holds in QOS-CARE 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* (cf. Def. 4.11), we first perform the reconfiguration on a graph model of the actual managed application and obtain a reconfiguration plan (cf. Def. 4.7). 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. Thus, if the reconfiguration process performed in our model succeeds, we assume that it finishes satisfactorily instrumented in the managed application (as assumed in the robustness property). Otherwise, if any of the previous steps fails, the actual managed application is not instrumented with the reconfiguration plan, thus left unmodified.

Therefore, given that QOS-CARE performs a reconfiguration in the actual running managed application only if all of its steps succeeds, it is easy to conclude that QOS-CARE guarantees the atomicity of the reconfiguration process.

6.8 Chapter Summary

In this chapter we have presented the validation of the atomicity and robustness properties, the conditions to verify the reconfiguration termination and SCA structural conformance, and how to measure and determine the acceptability of the settling-time. These are the five adaptation properties we used to define the reliability of our formal model for preserving QoS contracts, its derived SCA architecture, and respective implementation.

To achieve this, we first analyzed the classic definitions of reliability and identified that self-adaptive software systems already address some of the concerns implied by these definitions; conversely, these definitions omit other important considerations that must be addressed when evaluating the reliability of this kind of systems. For instance, one of these considerations is that natural context changes must be included as another possible cause for disrupting the continuity of agreed services delivery. Thus, reliability definitions require to be refined and reconsidered in order to be applied to self-reconfiguring software systems. Second, based on this analysis, and the adaptation properties defined in Chapter 3, we selected five of these properties that consistently fulfill the considerations required for assessing the reliability of this kind of systems.

In light of this, the motivation for developing a formal model gains more relevance. We needed a formal basis to guarantee desirable properties on the reconfiguration mechanism, as far as possible, or at least, to provide it with a formalism that admits formal analysis on these properties. In our opinion, it is not only important to develop reconfiguration mechanisms based on novel adaptation strategies, but also that these mechanisms be guaranteed in terms of standard and comparable properties. In the case of this dissertation, we guarantee the reliability of the software reconfiguration process to preserve QoS contracts by means of validating and verifying the enunciated adaptation properties. We also classified these properties based on *when* (i.e., design-time vs. run-time) and *where* (i.e., reconfiguration mechanism vs. managed software application) can they be validated and verified.

In the next chapter, we complement the validation and verification of these properties by using QOS-CARE to preserve QoS contracts in two different application scenarios. In particular, we analyze the QOS-CARE mean-time to reconfigure (MTTR) for each of these two application scenarios, and verify the respective conditions for termination and full-structural conformance on their reconfiguration rule-sets.

Chapter 7

QoS-CARE Validation Scenarios

Contents

In the previous chapter we presented the validation and verification of QoS-CARE's formal properties. In this chapter we complement this validation and verification by applying QoS-CARE to preserve QoS contracts in two different application scenarios, and executing them with the FraSCAti runtime platform. We also evaluate the applicability and performance (i.e., the mean-time to reconfigure, MTTR) of our formal model and its realization by executing a benchmark defined as a set of experiments on these two application scenarios.

The first validation scenario corresponds to a completed version of the example used through this document to illustrate the formal definitions of our reconfiguration mechanism. This is a mobile videoconference system subject to a QoS contract guaranteeing the *availability* and *confidentiality* of the videoconference services under different context situations. The second is a simple Web mashup application that dynamically composes and orchestrates the location service of the Twitter[30] social network with a weather web service from different available weather information providers, such as Google[31], Yahoo[32], and WebServiceX[33]. Based on the satisfaction

[30]`https://dev.twitter.com/docs/api`
[31]`http://code.google.com/p/java-weather-api`
[32]`http://weather.yahooapis.com/forecastrss`
[33]`http://www.webservicex.net/ws/WSDetails.aspx?CATID=12&WSID=56`

of a contract on the QoS property of *readiness* (cf. [Avizienis *et al.*, 2004]) on the weather service, the *availability* of the `weather-for-a-twitter-user` service application is guaranteed.

For each of these scenarios, we illustrate the QoS contract requirements to be satisfied by the respective component-based software applications, and present the experimental measurements gathered from their execution. For the particular reconfiguration rule-sets we analyze the termination and SCA structural conformance conditions, whereas for the experimental results, the mean-time to reconfigure as well as the QOS-CARE overhead in the corresponding SCA runtime system. In other words, we also complete our analysis and validation of QOS-CARE's reliability as defined in the previous chapter, that is, in terms of the adaptation properties of atomicity, robustness, short settling-time, termination, and CB-structural conformance. Nonetheless, we focus our analysis on the last three properties, recalling from the previous chapter that for the first two we gave sufficient conditions that we proved as a result of our formal model. Additionally, we incorporated the conditions necessary to verify the full CB-structural conformance at runtime as part of the *reconfig* operation, while we can verify the ones for termination at loading time. For this, we benefit from the respective AGG graph-transformation component functionalities.

The importance of this experimental evaluation, besides illustrating how to complete the validation and verification of QOS-CARE properties for particular applications, is that its results allow us to confirm the practical feasibility, applicability, and (re)usability of our reconfiguration mechanism and its formal model. Moreover, the application scenarios illustrate the potentialities of QOS-CARE as a complement to SCA and Service Oriented Computing (SOC) platforms to enable service component applications to satisfy their QoS contracts.

This chapter is organized as follows. In Section 7.1 we describe the hardware and operating software systems configuration used for executing the application scenarios and evaluation experiments. In Sections 7.2 and 7.3 we respectively present each of the QOS-CARE application scenarios, analyzing the termination and SCA structural conformance conditions, and the experimental data on the mean-time to reconfigure. In Section 7.4 we analyze and discuss limitations of QOS-CARE in contrast to its advantages as an SCA layer for preserving QoS contracts in component runtime platforms. Finally, we conclude with Section 7.5.

Correspondences in this Chapter: *Addressed Challenge(s):* C5 –The realization of the reconfiguration mechanism for preserving QoS contracts must be feasible as a software architecture and implementation. This implementation must be executable by existing component runtime platforms with reasonable performance. *Goal(s):* G5 –Determine the practical feasibility of the formal based reconfiguration mechanism to preserve QoS contracts. *General contribution(s):* GC.AIE –Formal model (GC.FM1 and GC.FM2) realized and evaluated as an SCA layer for dynamic reconfiguration. *Specific contribution(s):* AIE.1; AIE.2 –SCA layer architecture for dynamic reconfiguration to preserve QoS contracts designed and implemented maintaining the formal model properties. Formal model's proof-of-concept implementation experimentally evaluated in a real SCA platform; practical feasibility and (re)usability of reconfiguration mechanism determined.

7.1 General Platform Configuration for Executing the Validation Scenarios

To execute QOS-CARE with its validation scenarios and evaluation experiments we set several platform configurations. The hardware and software used in these configurations (single and multi-machines) was based on Intel i3@2.4Ghz processors with 4Gb of RAM running GNU/Linux Fedora 15 with non-relevant services and applications shut down. For the SCA platform we used FraSCAti 1.4 with Java 1.6.0_23 allocated with 128MB of RAM.

The main purpose of these configurations is, of course, to measure the performance of our reconfiguration mechanism, and determine its suitability and adequacy as an additional layer to preserve QoS contracts in SCA middleware stacks. We measure this performance in terms of the reconfiguration settling-time property, and the added overhead caused by this additional SCA layer in the component runtime system. To obtain the respective measurements we designed a small benchmark consisting of several experiments, and repeated them 10,000 times on each of the application scenarios, as we illustrate in the following sections.

7.2 Application Scenario 1: A Reliable Mobile Videoconference System

Trough this dissertation we have used a reliable mobile videoconference software system (RVCS) to illustrate the definitions of our formal model, as follows:

i. In Section 2.5, p. 40, we described the RVCS application scenario and its requirements. The RVCS system manages videoconference meetings and provides services to users for registering and virtually attending to these meetings. As confidentiality is a concern for this system, connections from the intranet are considered secure, thus clear communication channels can be used. From the extranet, confidential channels are required to be configured, whereas in case of no connection, a local cache structure. With this application scenario we also illustrated the challenges that the uninterrupted satisfaction of QoS contracts implies for self-reconfiguring software systems.

ii. In Section 4.2.2, p. 69, we presented a simplified component-based software application that satisfies the given requirements, both as an SCA software architecture and also represented as an e-graph.

iii. In Sections 4.2.3, p. 71, and 4.3.1, p. 77, we explained the QoS contract specification that regulates the provision and delivery of the RVCS services, and the corresponding formal semantics of this contract, respectively.

iv. In Section 4.2.4, p. 73, we presented a reconfiguration rule from the *R_confidentChannel* rule-set in the specified QoS contract.

v. In Example 4.4, p. 76, we illustrated the corresponding reconfigured software system that results from the application of this rule-set.

In the following sections we complete the reconfiguration rule-set for the given QoS contract, and analyze the conditions necessary to verify the full CB-structural conformance, the reconfiguration termination, and determine the acceptability of the reconfiguration settling-time for this application scenario.

7.2.1 Reconfiguration Rules

As presented previously, in this contract we have three QoS levels corresponding to context conditions defined by the network access point used by the software client at every moment of its execution, that is, if the user is connected from an intranet-serviced area, an extranet-serviced area, or has no network access at all. 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

[Ramachandran, 2002, Dougherty *et al.*, 2009]: a clear channel when connected from the intranet; a ciphered channel when from the extranet; and a local cache when having no network access. We present these reconfiguration rule-sets in the following sections.

Rules for Changing to an Intranet-Serviced Area

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 Fig. 7.1 we present the rule-set to apply when the user moves to an intranet-serviced area from both, an extranet-serviced area (cf. `extra2intra` rule in the top of the figure) and an area having no network access at all (cf. `nonet2intra` rule in the bottom of the figure).

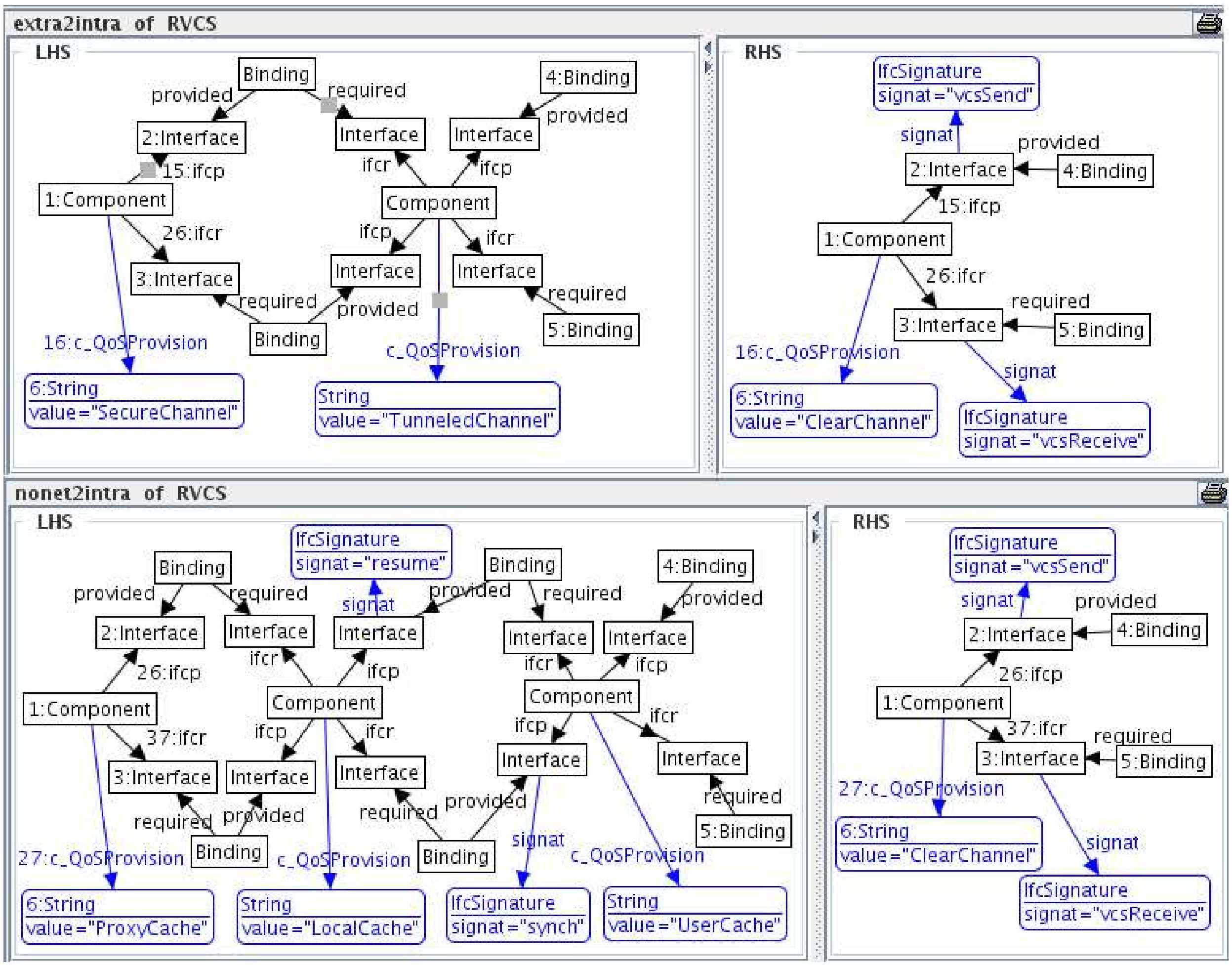


Figure 7.1: The *R_clearChannel* reconfiguration rule-set.

The local cache design-pattern we used 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 these figures for legibility reasons.

Rules for Changing to Extranet-Serviced and Unreachable-Network Areas

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 and unreachable-network areas (i.e., *R_confidentChannel* and *R_localCache*) can be specified from the previously specified rules, by inverting and exchanging their LHSs and RHSs, that is, with LHS = RHS = $\{intranet, extranet, noNetwork\}$, as arranged in Fig. 7.2 (e.g., the rule `extranet-to-intranet` is formed with LHS = $extranet$ and RHS = $intranet$).

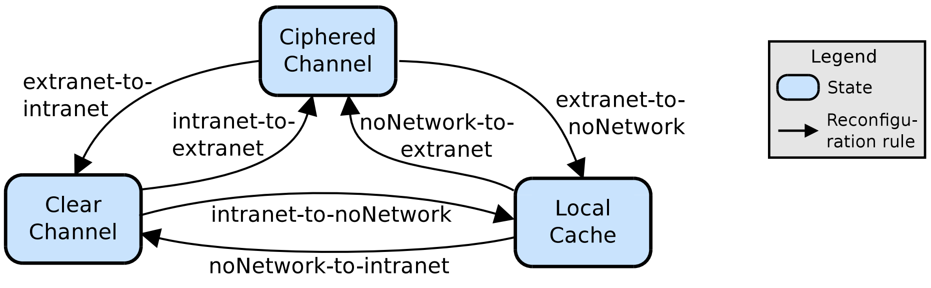


Figure 7.2: The reconfiguration rule-sets for the confidentiality contract.

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 (i.e., two reconfiguration rules).

7.2.2 Runtime Verification of Reconfiguration Properties

Component-Based Structural Conformance

As explained in previous chapters, we incorporated the verification of the full CB-structural conformance conditions specified in Section 6.6 in the *reconfig* operation. Thus, this verification is performed at runtime after every reconfiguration of the component-based managed application. Nevertheless, we can verify these conditions by analyzing the reconfiguration rules illustrated previously for this application scenario, as follows.

i. The reconfiguration rule-sets *R_clearChannel*, *R_confidentChannel*, and *R_localCache* are built from the LHSs and RHSs specified in the previous section, that is, with LHS = RHS = $\{intranet, extranet, noNetwork\}$. Thus, it is sufficient to analyze the full CB-structural conformance conditions on these e-graphs.

ii. All of these e-graphs share the same key elements for the reconfiguration operation, and these elements are used consistently in all of the LHSs and RHSs. These elements are the `2:Interface`, `3:Interface`, `4:Binding` and `5:Binding`, and they invariably enclose the intermediate elements to be reconfigured.

iii. All of the reconfiguration rules either add or delete connected elements that are enclosed by these key elements.

iv. As far as we apply these rules to full CB-structural conforming structures, it is easy to verify that the performed reconfigurations do not alter this property on them.

Reconfiguration Termination

In Section 6.4 we showed that our formal model for dynamic reconfiguration can be reduced to a typed attributed graph transformation system (TAGTS). Hence, our formal model inherits the TAGTS formal properties, including the termination conditions for layered graph transformation systems (LGTS). The Attributed Graph Grammar system (AGG) implements this layered strategy of graph transformation, being this strategy a generalization of our *QoSC_FSM* $\rho : STATES \rightarrow \mathcal{P}(\Gamma)$ auxiliary function. The reconfiguration rule-sets *R_clearChannel*, *R_confidentChannel*, and *R_localCache* are just particular names for this function, applied to the respective states.

While we use ρ to apply only the reconfiguration rule-set that is specific to fulfill the QoS level corresponding to the target state to be reached under the changed context condition, we use AGG's LGTS to determine whether the rules in a given rule-set must be grouped to be applied in different layers. AGG not only incorporates the enunciated termination conditions in the respective verification on the reconfiguration rule-sets, but also generates the layers to re-group the reconfiguration rules, if necessary.

Thus, for the verification of the termination conditions specified in Section 6.4 for the given reconfiguration rule-sets of this application scenario, we use the corresponding functionality of the AGG system on each of these rule-sets. As illustrated in Fig. 7.3, the two rules in the reconfiguration rule-set *R_clearChannel*, to be applied when changing to an intranet-serviced area, satisfy the termination conditions and can be used in the same layer. The analysis also illustrates the elements of the rules that present conflicts as classified in the creation and deletion layers (middle and bottom parts of the figure, respectively). In this case, no termination conflicts were detected, as all elements could be classified in the same layer.

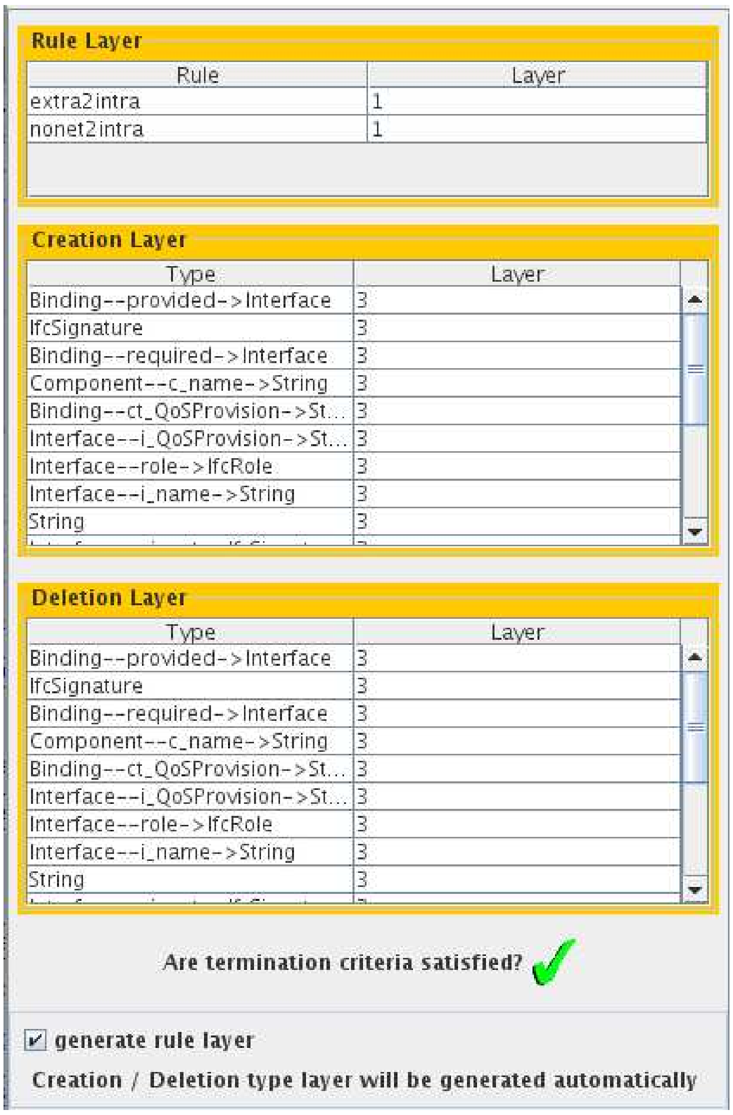
We performed the same analysis for the two other reconfiguration rule-sets, *R_confidentChannel* and *R_localCache*, with the same results.

Settling Time

Compared to the other four, the settling-time is a different kind of property because it must be analyzed and measured completely by experimentation at runtime.

To obtain the MTTR measurement as accurately as possible, it must be performed under controlled situations, that is, without any disturbances (e.g., with no added CPU load of other processes, and no superfluous operating system services), and repeated several times under similar conditions. To achieve this, we followed the general conditions established in Section 7.1, and executed every experiment by repeating the evaluated operation 10,000 times. Then, we averaged the results in the same evaluation program. We repeated these 10,000-times experiments several times, under the same conditions, that is, only varying the day and time of execution, obtaining consistent results (i.e., differences of 2-3 milliseconds among the averages).

In addition, to evaluate the reconfiguration settling-time for this particular application scenario, we designed a benchmark with different configurations and evaluation objectives. To obtain basic measurements, the configurations comprise only one client vs. the videoconference server. As illustrated in Table 7.1, the benchmark focuses on the mean-time to reconfigure several RVCS deployment variations, that is, either executing the client and server in the same machine,


Rule Layer

Rule	Layer
extra2intra	1
nonet2intra	1

Creation Layer

Type	Layer
Binding--provided->Interface	3
IfcSignature	3
Binding--required->Interface	3
Component--c_name->String	3
Binding--ct_QoSProvision->St...	3
Interface--i_QoSProvision->St...	3
Interface--role->IfcRole	3
Interface--i_name->String	3
String	3

Deletion Layer

Type	Layer
Binding--provided->Interface	3
IfcSignature	3
Binding--required->Interface	3
Component--c_name->String	3
Binding--ct_QoSProvision->St...	3
Interface--i_QoSProvision->St...	3
Interface--role->IfcRole	3
Interface--i_name->String	3
String	3

Are termination criteria satisfied? ✓

☑ generate rule layer

Creation / Deletion type layer will be generated automatically

Figure 7.3: Termination analysis for the *R_clearChannel* reconfiguration rule-set.

or in different machines; and thus, communicating them either directly, or over a loopback, or yet over a 11-hops Internet connection. The measured times were taken from the execution of the reconfiguration mechanism in the client with the reconfiguration rules presented previously.

The first row of the table presents the time for the most expensive SCA primitive reconfiguration operation: loading a component. Naturally, this operation depends on the size of the component, and the operation required to deploy the component bytecode and composite in the target machine (e.g., if the component files are stored locally or they must be transmitted over an Internet connection).

The second row indicates that applying a one-rule reconfiguration in the e-graph representation and the verification of full SCA conformance conditions takes 68ms. Rows 3 to 5 illustrate the total MTTR for reconfiguring the RVCS's clear channel configuration into a ciphered channel one. Using a local (i.e., direct) interface takes 624ms., whereas 640ms. for a remote REST interface over a loop-back in the same machine, and 876ms. instrumenting the reconfiguration over an Internet connection. The difference between the reconfiguration at the e-graph level vs. the SCA level is that in the latter QoS-CARE must interact with the actual software application,

Table 7.1: RVCS settling-time benchmark scenarios and results

Evaluation Objective in Configured Scenario	Time (msec.)
Local component loading/unloading (5395 bytes)[a]	41
E-Graph transformation (one rule application[b])	68
Local reconfiguration (total MTTR local interface)[a]	624
"Remote" reconfiguration (total MTTR over loopback)[a]	640
Remote reconfiguration (total MTTR over Internet)[c]	876
QoS-CARE overhead (simulating 1 dummy event/3sec.)[a]	3

[a] One client vs. server in the same machine.
[b] Rule for changing to a "confident channel" from a "clear channel" configuration.
[c] Instrumented over a ~4Mbps, 11 hops Internet connection.

while it is *in execution*. Thus, instrumenting the reconfiguration plan at the SCA level requires to stop/re-start the execution of the intervened components before/after performing the reconfiguration changes. Finally, the measured overhead of QoS-CARE in this application scenario is of 3ms.

To take the measured times, we used FraSCAti to execute QoS-CARE and the RVCS system from its initial configuration, as explained in Chapter 4. We configured the *Context Events Simulator* component of QoS-CARE to generate simulated reconfiguration events notifying changes in the user's network access, from intranet- to extranet-serviced areas. To obtain measurable times, a second event is simulated for QoS-CARE to apply the inverse of the same rule. These two events are then cyclicly repeated for the desired number of times.

The reconfiguration script generated by the *Planner* component to be applied for changing to the confident channel involves the following steps:

i. deploy and load the *EnDeCipher* component for enciphering/deciphering the transmitted messages, once the user moves from the intranet to the extranet;
ii. add it into the client composite;
iii. un-promote the client reference to the chat server service;
iv. re-wire the client reference to the *EnDeCipher* component; and
v. promote the *EnDeCipher* reference to the chat server service in the client composite.

Qualitative Analysis

In our opinion, the measured MTTRs (e.g., the settling-time of 876ms. over an Internet connection) are acceptable enough, compared to existing mobile and remote (e.g., over Internet) services latency. Moreover, considering that the confidentiality is guaranteed while ensuring the continuity of the service at the negligible overhead of 3ms., this settling-time is a very affordable cost in time.

In light of this, the obtained MTTRs for this application scenario confirm the practical feasibility of using QoS-CARE as a platform-independent SCA layer for preserving QoS contracts in component-based software applications through dynamic reconfiguration. In addition to the possibility of being used in any SCA platform, QoS-CARE enforces the separation of concerns between the managed application and the reconfiguration mechanism, and their corresponding properties. First, the QoS contracts are specified independently of the managed application. Second, except for the monitor probes, the managed application does not require of any modification for the reconfiguration mechanism to make it preserve its QoS contract. Third, the

MAPE model elements remain explicit in the QoS-CARE's SCA architecture, which enables not only their distribution in several machines, but also facilitates their replacement. Finally, this independence and separation of concerns allow us to analyze the reconfiguration mechanism's adaptation properties. In particular, we have successfully completed the validation and verification of five of these adaptation properties (i.e., properties inherent to self-adaptive software) in QoS-CARE.

7.2.3 Implementation Details

In this section we present some implementation details of the analyzed application example in terms of its components, classes, physical lines of code (LOC), and density (i.e., LOCs per class). This application example is based on a plain old Java object (POJO) implementation of a videoconference system developed independently of this dissertation[34], following the respective requirements introduced in Section 2.5. As summarized in Table 7.2, the application example is composed of 7 SCA components, which are implemented with 9 Java interfaces and 140 Java classes, and reusing the monitor interface from QoS-CARE.

Table 7.2: RVCS application example implementation details

RVCS Application Example	Components	Classes and Interfaces	LOC	LOC/ Class
Monitoring Elements	1	1	84	84
Application Logic[a]	6	148	23,799	160
Total	7	149	23,883	160

[a] Reconfiguration applied only to the use cases subject to the QoS contract on confidentiality.

To preserve the QoS contract on confidentiality in this application with QoS-CARE, we needed first to group and encapsulate its Java classes in SCA components, and publish its functionalities as RMI and Web services. Then, for the specific tasks for preserving the contract, we developed:

- One monitoring component (84 LOC), to notify QoS-CARE about context changes in the network access (i.e., from intranet, from extranet, and no network), signaling an imminent violation of the contracted QoS levels.
- 6 application-logic components (116 LOC for packaging the POJO classes in 3 composite files).

Hence, apart from the QoS contract specification (including the 6 reconfiguration rules), the net development work for preserving the contract on the application example amounts to the monitoring component, that is, 84 LOC (i.e., 0.4% of the total). This represents a very low amount of work compared to the total application size, considering that all of our remaining work was devoted to enable the application to be executed in an SCA runtime platform (i.e., in FRASCATI). Naturally, integrating the monitor probe in the managed application required to understand the application code (in this case, in the network adapter component), even though this monitoring could be performed at the operating system level.

[34] http://gforge.icesi.edu.co/gf/project/seams

7.3 Application Scenario 2: A Dynamic Twitter-Weather Mashup

This application scenario illustrates how QoS-CARE can be used to create a `weather-for-a-twitter-user` service as a web mashup application. This application dynamically composes and orchestrates the location service of a Twitter user (i.e., the city/country as stored in the user's profile) with a weather web service from different available weather information providers, such as Google, Yahoo, and WebServiceX. The use of QoS-CARE to preserve a contract on the QoS property of *readiness* on the weather service serves as a guarantee for the availability of the mashup application. As characterized by [Avizienis *et al.*, 2004], service readiness measures immediate availability in the sense of immediate response. In contrast, continuous availability (or simply, availability) measures the continuity of a service uninterruptedly, after it has initially (and partially) answered service requests.

That is, the idea of the QoS contract is to guarantee, for the service user, the weather conditions report from any of the weather information sources that she has registered in the application or in the contract, and shows its readiness to provide it. Moreover, these weather information sources can be added at runtime as contract re-negotiations.

7.3.1 Component-Based Application Structure

The component-based application structure of our solution is partially based on two FraSCAti examples that provide components and services for two basic functionalities[35]:

- twitter: for a given user, retrieves and decodes all of her public profile information. This includes her registered city and country.
- weather: retrieves the weather conditions on a given location as a pair city-country, using the WSDL weather information service from `http://www.webservicex.net/globalweather.asmx`.

For our solution, we modified and reused the core components of these two examples to compose their mentioned services as illustrated in Fig. 7.4. For a given user (the `userId` exposed property in the figure), the Twitter-Weather Mashup component (`TWMashup`) requests the user profile from Twitter, and uses the XML Twitter profile decoder component (`Decoder`) to obtain the registered location as a city-country pair. Given that existing weather information providers have different interfaces and method signatures to deliver the weather information, we also replaced the weather service interfaces with generic ones for using the existing services independently of their particular definitions. The Weather Orchestrator component (`WeatherOrchestrator`) is responsible for providing the generic weather service (through the provided interface named `WeatherSCAService`), initially only from the Internet WebServiceX provider, through the `WSXWeather` component. This component is responsible for translating the generic weather request and respective response to the particular WebServiceX interface specification and requirements.

The corresponding component-based structure application reflection (CBSAR) defined in e-graphs is illustrated in Fig. 7.5, omitting the `Decoder` component, the interface names, and the Internet service bindings for legibility reasons.

[35] `http://websvn.ow2.org/listing.php?repname=frascati&path=%2Ftags%2Ffrascati%2Ffrascati-1.4%2Fexamples%2Ftwitter` and `http://websvn.ow2.org/listing.php?repname=frascati&path=%2Ftags%2Ffrascati%2Ffrascati-1.4%2Fexamples%2Fweather`

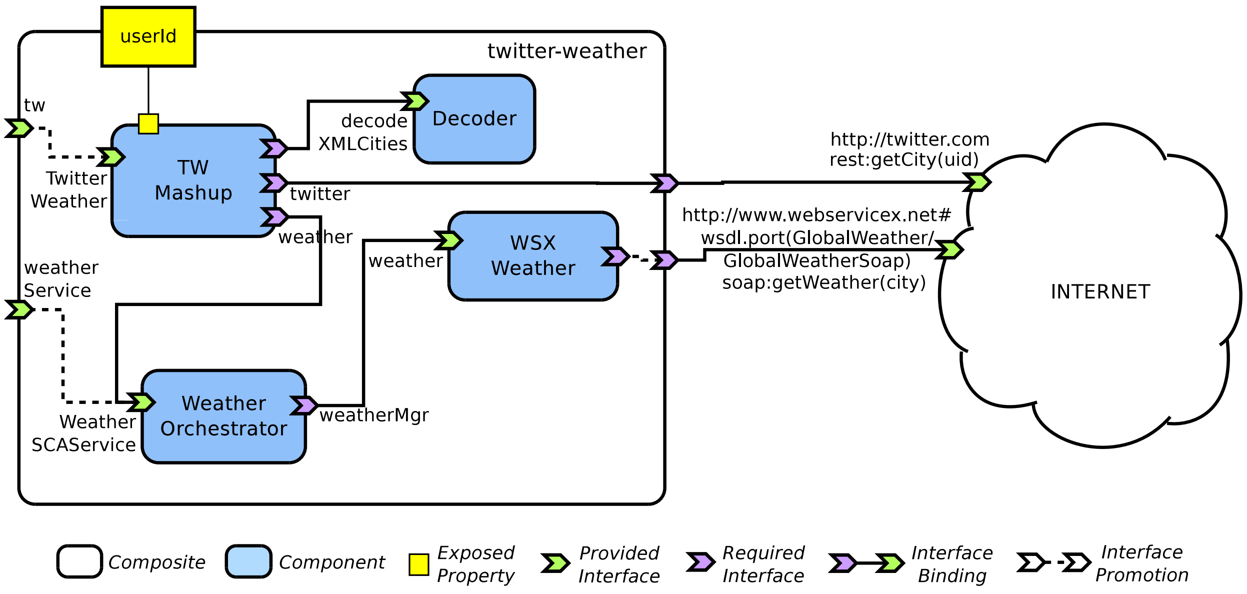


Figure 7.4: The Twitter-Weather SCA application.

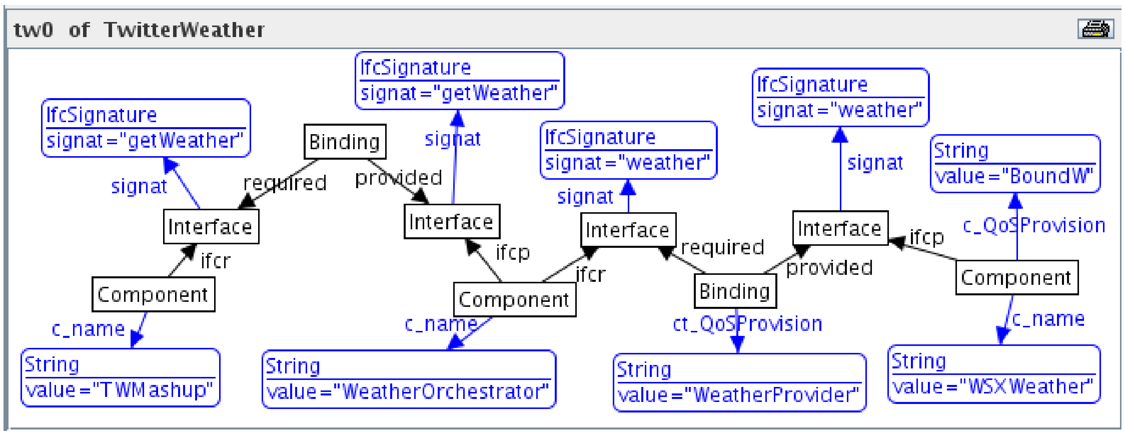

tw0 of TwitterWeather
IfcSignature signat="getWeather"
IfcSignature signat="getWeather"
IfcSignature signat="weather"
IfcSignature signat="weather"
String value="BoundW"
Binding
signat
required
provided
Interface
ifcr
ifcp
Component
c_name
String value="TWMashup"
String value="WeatherOrchestrator"
String value="WeatherProvider"
String value="WSXWeather"
ct_QoSProvision
c_QoSProvision

Figure 7.5: The Twitter-Weather CBSAR structure in e-graphs.

7.3.2 QoS Contract

Public web-services for providing weather information are massively used for different purposes, being it common to find them as non-available, or more precisely, unready to respond weather requests. To cope with this problem, we define a contract on the readiness of weather service providers, as illustrated in Table 7.3. On one side, line 1 in this table specifies that whenever occurs the context event *weatherProviderTimeOut* (e.g., the current weather information provider presents a service unreadiness), QOS-CARE must apply the *R_ChangeWeatherProvider* reconfiguration rules. These rules change the weather information provider in order to fulfill the corresponding QoS level. On the other side, line 2 offers the possibility of having rules to add new weather information providers, triggered by the *newWeatherProviders* event (e.g., as a result of a contract renegotiation). The *R_AddWeatherProviders* rule-set deploy the required additional components to use the new weather information providers at runtime. Of course, in this case the user must supply the corresponding component implementations for translating the weather request/response to/from these new providers to be able to use them with the respective reconfiguration rules. At runtime, these components are dynamically deployed at the first time of the corresponding rule application.

Table 7.3: QoS contract on service readiness

Twitter-Weather Mashup Readiness Obligations		
Context Events	QoS-Level Objective	Guaranteeing Rule Set
1: $weatherProviderTimeOut$	$weatherProviderReady$	$R\_ChangeWeatherProvider$
2: $newWeatherProviders$	$weatherProvidersAdded$	$R\_AddWeatherProviders$
Assignment of Responsibilities		
- System Guarantor:	$System.WeatherOrchestrator$[a]	
- Context Monitor:	$System.WeatherTimeOutProbe$[b]	

[a] The application component responsible for resolving weather requests.
[b] The designated component for monitoring time-outs on requests to the external weather information providers.

For the weather service readiness, in this contract we opt for specifying only one QoS level to fulfill. Even though we could specify a more refined set of context conditions with their respective QoS levels and reconfiguration rule-sets to choose a weather information provider based on more complex criteria, we prefer to maintain the contract simple enough to illustrate other potentialities of QoS-CARE not yet explored as an SCA layer for driving the dynamic reconfiguration of component-based applications, as follows in the next sections.

7.3.3 Reconfiguration Rules

As presented previously, this contract comprises two reconfiguration rule-sets. The first is for performing the change of weather information provider among the registered ones, while the second for introducing new weather information providers. Although it is possible to perform both tasks in a same rule, this is not desirable. First, the rules for introducing new weather information providers require to be specific in the particular attributes that characterize the new weather service. Second, the rules for changing of weather information provider must be general enough to use any of the registered ones, independent of their particularities. Third, by separating the tasks in these two different kinds of rules we maintain the simplicity, modularity, and maintainability of each of them.

Rules for Adding New Weather Information Providers

The structure of rules for introducing new weather information providers is exemplified by the rule illustrated in Fig. 7.6, which adds the Google weather information provider. Given that the left-hand side (LHS) of the rule is empty (i.e., it holds in any CBSAR e-graph), the negative application condition (NAC) of the rule simply prevents of applying the rule more than one time. The right-hand side (RHS) of the rule introduces the components, interfaces and wiring required for accessing the service of the new weather information provider. We omit in this figure details such as the attribute specifying the Java implementation of the `GoogleWeather` component, for legibility reasons.

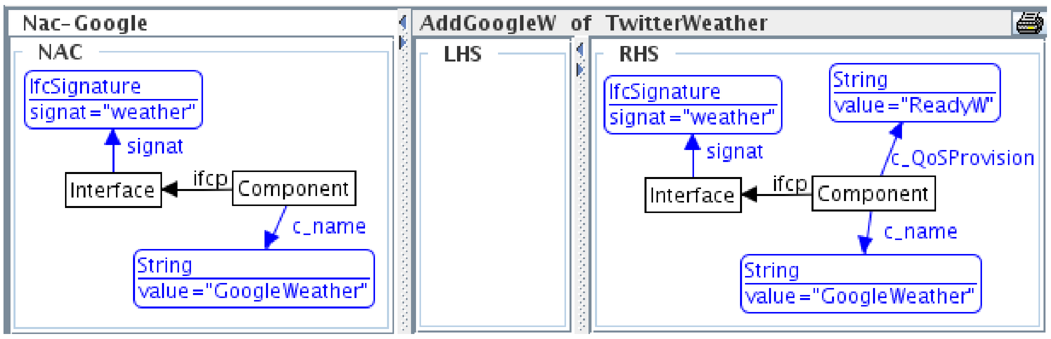


Figure 7.6: Reconfiguration rule for adding the `GoogleWeather` provider.

Rules for Changing of Weather Information Provider

For the reconfiguration rules that actually perform the change of weather information provider, we use simplified design patterns from [Buschmann *et al.*, 2007] that follow a strategy of resource redundancy to address availability problems. Assuming the same cost/benefit of using any of the registered providers, we use a non-deterministic choice among them. However, we split the strategy in two rules, and apply them in alternative rounds, one after the other.

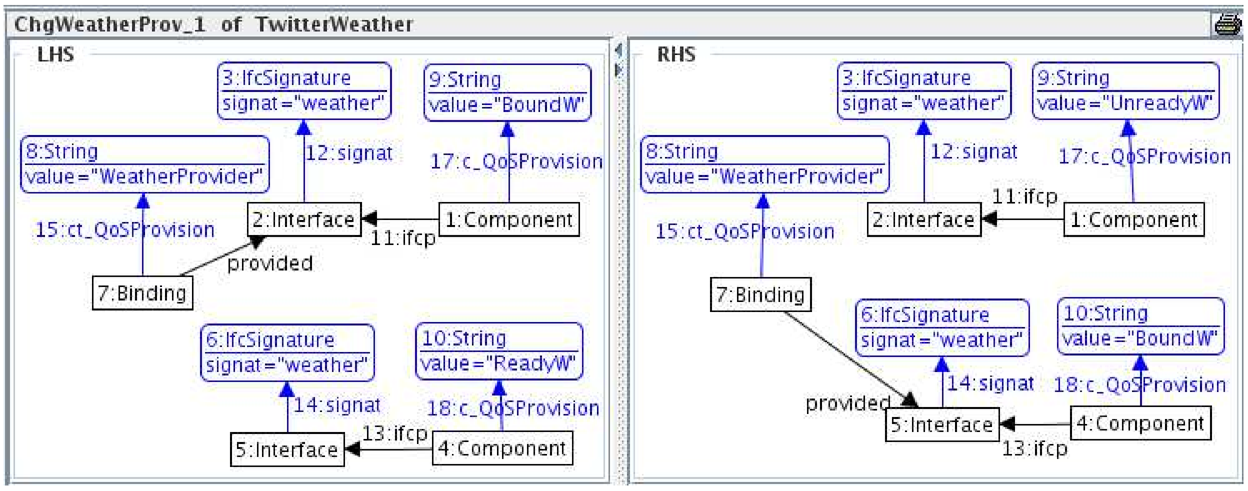


Figure 7.7: Reconfiguration rule for changing to a "ready" weather information provider.

The first rule, illustrated in Fig. 7.7, tries to find a match of its LHS in the managed application for the following elements:

i. the binding responsible for provisioning the weather information provider, that is, the binding with index 7[36]. In the figure, this is `7:Binding` having attribute `15:ct_QoSProvision` with value `"WeatherProvider"`. This binding is bound to the provided `weather` interface `2:Interface` of `1:Component`, whose attribute `17:c_QoSProvision` thus has value `"BoundW"`; and

ii. an existing component for accessing a "ready" weather information provider, that is, the component with index 4 (cf. `4:Component` thus having attribute `18:c_QoSProvision` with value `"ReadyW"`, and a provided `weather` interface `5:Interface` in the figure).

[36]Recall that these indexes are used to associate LHS with RHS elements; they are not used for the match operations.

If this match is found, the RHS of the rule re-binds the `provided` attribute of the identified `7:Binding` from interface `2:Interface` of the `"BoundW"` weather information provider component, to the `5:Interface` of the `"ReadyW"` one. Additionally, the rule marks the previous `"BoundW"` as `"UnreadyW"`, and updates the previous `"ReadyW"` as the currently `"BoundW"`. This means that the previous weather provider, given that it did not respond to a request, thus is marked as unready.

The second rule, illustrated in Fig. 7.8, performs a similar reconfiguration but interchanging the matching `"ReadyW"` by `"UnreadyW"` in the LHSs, and vice versa in the RHSs. This means that this rule retries on weather information providers that were marked as unready by the previous rule, in the previous round.

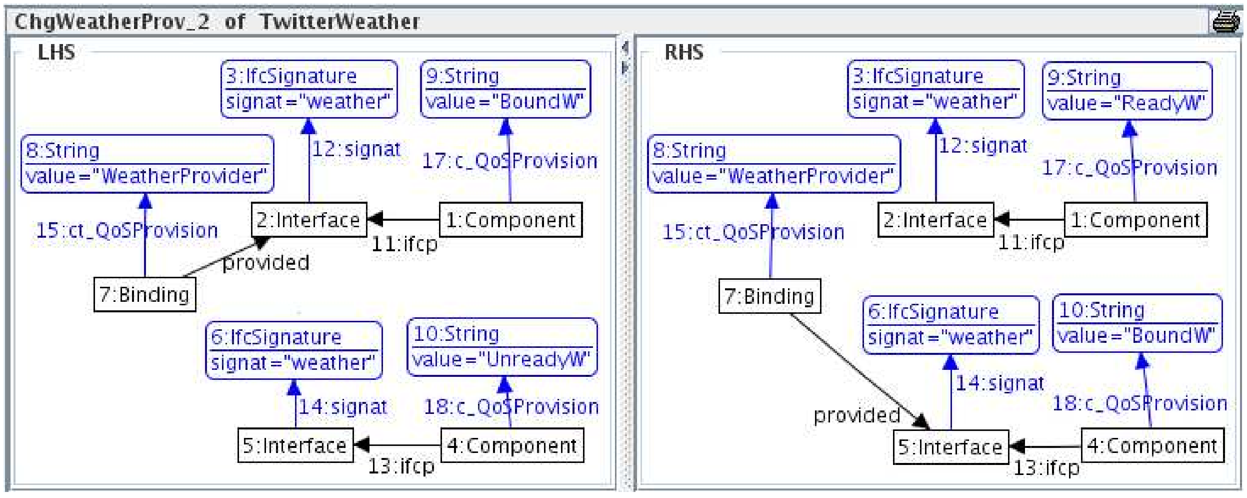


Figure 7.8: Reconfiguration rule for trying a previously unready weather information provider.

Generated Script for Changing of Weather Information Provider

From the application of the illustrated reconfiguration rule for changing to the Google weather information provider, a corresponding reconfiguration plan is generated. The `SCAInstrumentation` component (cf. Section 5.6.1, p. 101) translates this reconfiguration plan into the FScript script illustrated in Algorithm 7.1 (the names of the variables are modified for readability).

Algorithm 7.1 FScript script for changing to the Google weather information provider.

```
 1: orchestrator = $domain/scadescendant::WeatherOrchestrator;
 2: weatherRef = $orchestrator/scareference::weatherMngr;
 3: wProviderComp = $domain/scadescendant::GoogleWeather;
 4: wProviderSvc = $wProviderComp/scaservice::weather;
 5: wsxw = $domain/scadescendant::WSXWeather;
 6: wsxwSvc = $wsxw/scaservice::weather;
 7: set-state($orchestrator, "STOPPED");
 8: remove-scawire($weatherRef,$wsxwSvc);
 9: add-scawire($weatherRef,$wProviderSvc);
10: set-state($orchestrator, "STARTED");
```

In this script, the FraSCAti's FPath navigation language is used to retrieve the SCA managed application components. The meta-variable `$domain` in line 1 serves as a reference to the SCA contained composites and components managed by FRASCATI. `scadescendant`, `scaservice` and `scareference` are navigation language elements used to retrieve the respective SCA elements from these components. Lines 1-2 define the variables `orchestrator` and `weatherRef` as the `WeatherOrchestrator` component and its `weatherMngr` required interface, respectively. Analogously, lines 3-6 define the corresponding variables for the current weather provider (`GoogleWeather`) component and its `weather` provided interface, as well as the next "ready" weather provider (`WSXWeather`). Line 7 deactivates the `WeatherOrchestrator` component to perform the change of provider reference in lines 8 and 9. Finally, line 10 re-activates the `WeatherOrchestrator` component to resume its execution.

7.3.4 Reconfigured Application Structure

With the previously illustrated rules, QOS-CARE dynamically reconfigures the managed mashup application, not only to use different existing weather information providers, but also to add new ones, both at runtime.

Figures 7.9 and 7.10 respectively illustrate the SCA and CBSAR e-graph representations of the reconfigured mashup application after applying the rules for adding the Google and Yahoo weather information providers, as well as the change of corresponding provider. In this case, the Google weather information provider was chosen.

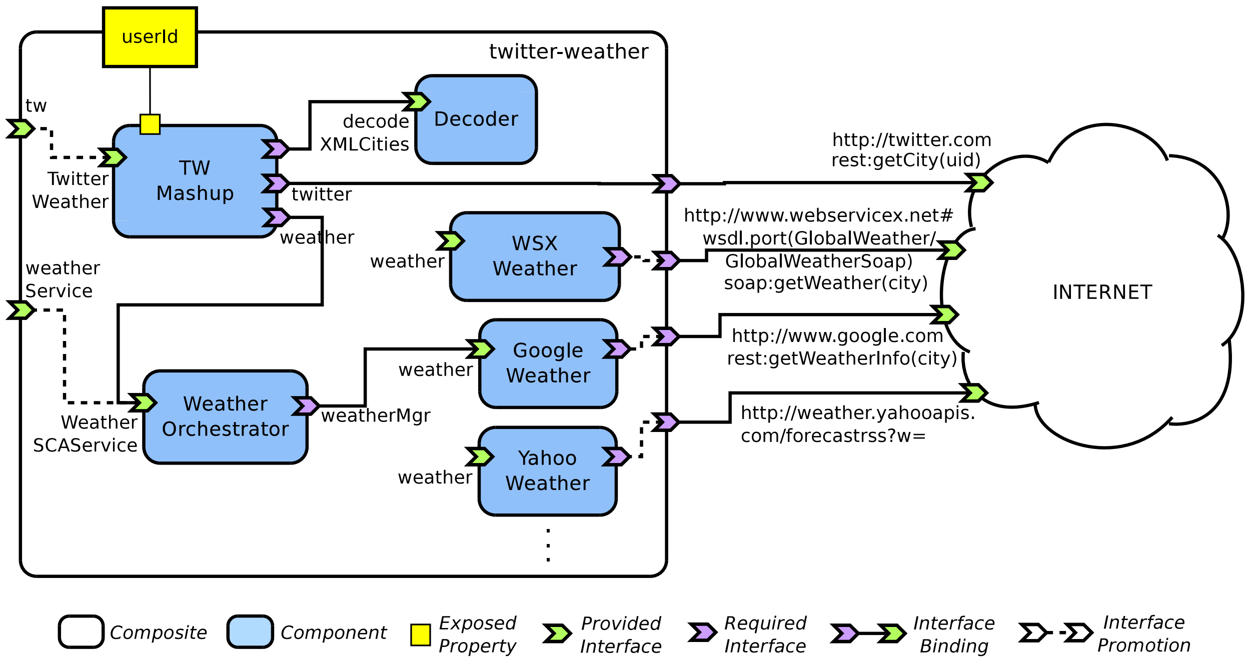


Figure 7.9: The Twitter-Weather SCA application reconfigured.

7.3.5 Runtime Verification of Reconfiguration Properties

Component-Based Structural Conformance

To verify the full CB-structural conformance conditions specified in Section 6.6 for the given reconfiguration rules of this application scenario, we observe the same remarks for the previous

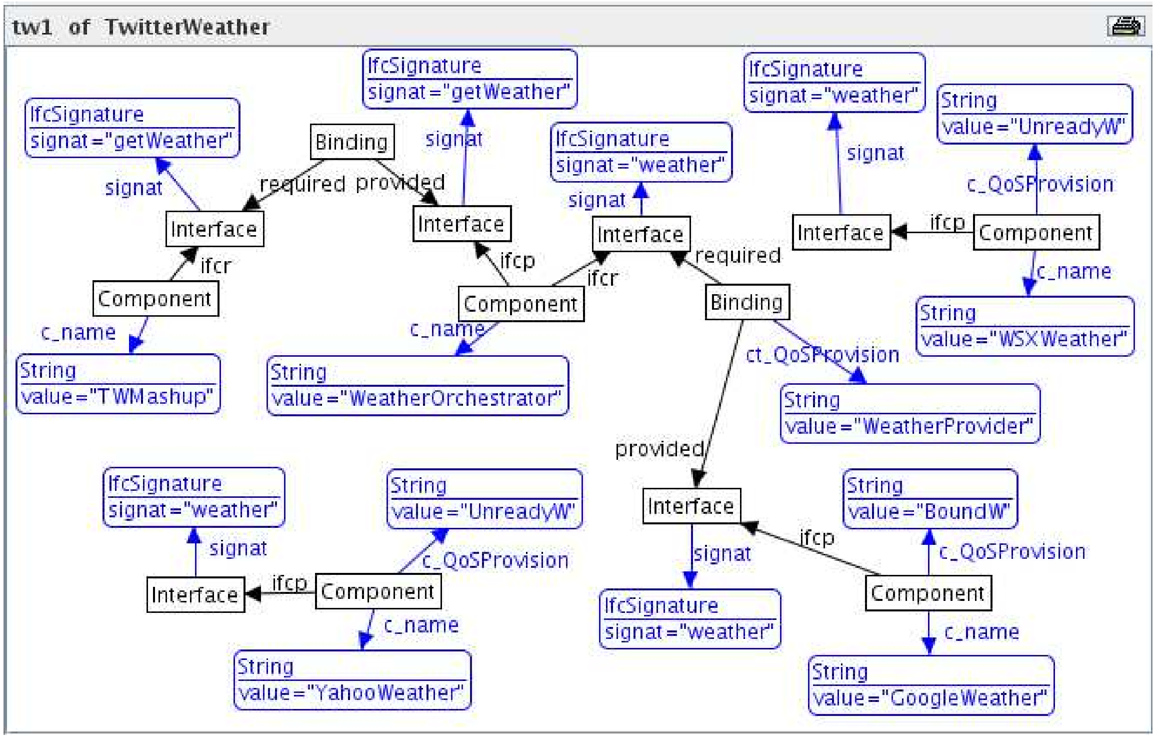

tw1 of TwitterWeather
IfcSignature signat="getWeather"
IfcSignature signat="getWeather"
IfcSignature signat="weather"
IfcSignature signat="weather"
String value="UnreadyW"
Binding
signat
required provided
Interface
Interface
Interface
Interface
Component
ifcr
ifcp
ifcr
required
ifcp
c_QoSProvision
c_name
Component
Component
Binding
String value="WSXWeather"
String value="TWMashup"
String value="WeatherOrchestrator"
ct_QoSProvision
String value="WeatherProvider"
provided
IfcSignature signat="weather"
String value="UnreadyW"
Interface
String value="BoundW"
Interface
Component
Component
IfcSignature signat="weather"
String value="YahooWeather"
String value="GoogleWeather"

Figure 7.10: The Twitter-Weather reconfigured CBSAR e-graph corresponding to Fig. 7.9.

application scenario (cf. Section 7.2.2). Additionally, we can verify these conditions by examining the reconfiguration rules illustrated previously for this application scenario, as follows.

i. For the rules to add new weather information providers, it is enough to verify that their right-hand side (RHS) structures satisfy, by themselves, the enunciated conditions. This is given that the respective left-hand sides are empty, and thus, no existing structures, already full CB-conforming, are modified.

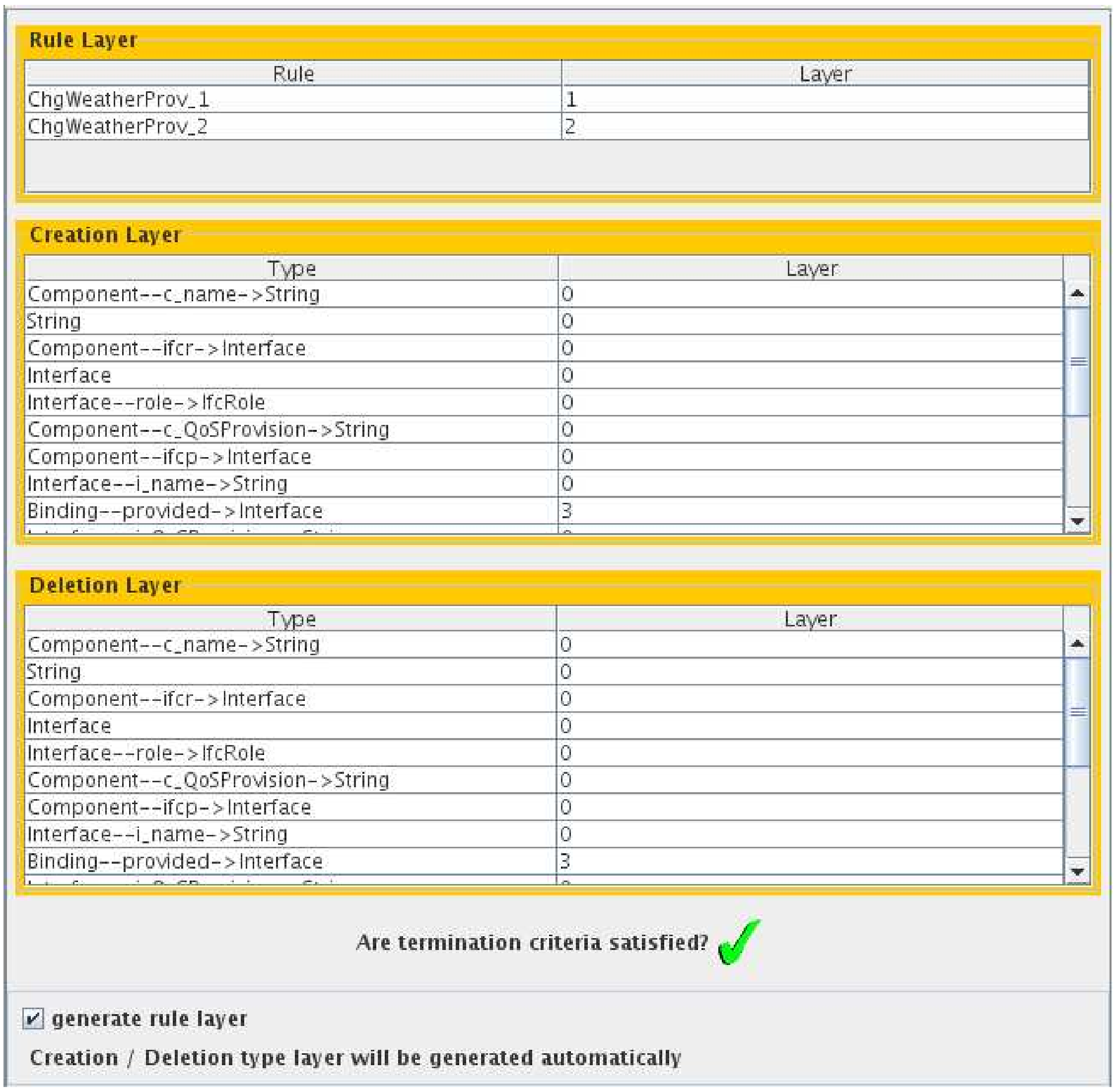
ii. Concerning the two rules that perform the change of weather information provider, the only structural modification introduced by the rule RHS is the rebinding from one interface to another, both of which have the same structure. These two rules are structurally equivalent, thus they do not alter the CB-conformance of the existing structures to be reconfigured.

Reconfiguration Termination

For the verification of the termination conditions specified in Section 6.4 for the given reconfiguration rules of this application scenario, we follow the same procedure used for the previous application scenario (see the discussion in Section 7.2.2).

That is, we use the layered graph transformation strategy (LGTS) used in AGG in conjunction with our *QoSC_FSM*'s $\rho : STATES \rightarrow \mathcal{P}(\Gamma)$ auxiliary function. We use ρ to apply the reconfiguration rule-set that is specific to change the weather information provider, and AGG's LGTS to verify the termination conditions on its two rules. If these conditions are not satisfied, AGG also generates the layers to re-group the reconfiguration rules. The resulting analysis is illustrated in Fig. 7.11.

According to this analysis, the rules `ChgWeatherProv_1` and `ChgWeatherProv_2` satisfy the termination conditions by separating them in two layers, 1 and 2 (cf. top part of the figure).



Rule Layer

Rule	Layer
ChgWeatherProv_1	1
ChgWeatherProv_2	2

Creation Layer

Type	Layer
Component--c_name->String	0
String	0
Component--ifcr->Interface	0
Interface	0
Interface--role->IfcRole	0
Component--c_QoSProvision->String	0
Component--ifcp->Interface	0
Interface--i_name->String	0
Binding--provided->Interface	3

Deletion Layer

Type	Layer
Component--c_name->String	0
String	0
Component--ifcr->Interface	0
Interface	0
Interface--role->IfcRole	0
Component--c_QoSProvision->String	0
Component--ifcp->Interface	0
Interface--i_name->String	0
Binding--provided->Interface	3

Are termination criteria satisfied? ✓

[✓] generate rule layer

Creation / Deletion type layer will be generated automatically

Figure 7.11: Reconfiguration termination analysis for the Twitter-Weather application scenario.

The analysis also illustrates the elements of the rules in conflict and that would be in the creation and deletion layers (middle and bottom parts of the figure, respectively). Comparing the two rules it is easy to observe that the conflicting element is the binding to the provided `weather` interface (marked as in layer 3 in the figure). This interface is deleted from the current weather information provider, and created for the ready one in both rules. The analysis indicating that it is necessary to separate the two rules is in agreement with our design decision to apply these rules in different rounds, which is equivalent to classify and apply them in different AGG layers, thus satisfying the termination conditions.

Settling Time

We disaggregate the benchmark for measuring the reconfiguration settling-time for the change of weather information provider as illustrated in Table 7.4. We took these average measurements from the execution of the previously illustrated reconfiguration rules, using the same experimentation criteria established for the previous application scenario (cf. settling-time in

Section 7.2.2).

The first row of the table indicates that the time for transforming the actual SCA representation, maintained by the FRASCATI runtime middleware, to the e-graph representation is 15ms. This is very similar to the time it takes for applying a one-rule reconfiguration in the e-graph representation and the verification of full SCA conformance conditions, 14ms. The script transmission over the loop-back TCP/IP stack is the most costly operation, with 81ms, while the actual execution of the FSCRIPT reconfiguration script takes 47ms. The measured overhead of QOS-CARE in this application scenario is the same as the measured in the videoconference scenario.

Table 7.4: Twitter-Weather mashup settling-time benchmark results

Evaluation Objective in Configured Scenario	Time (msec.)
E-Graph from FRASCATI's SCA domain translation	15
E-Graph transformation (one rule application[a])	14
FSCRIPT script transmission (473 bytes, 15 lines)[b]	81
FRASCATI's FSCRIPT reconfiguration execution	47
Total Mean-Time to Reconfigure (MTTR)[b]	157
QOS-CARE overhead (simulating 1 dummy event/3sec.)	3

[a] Rule for changing to a "ready" weather information provider.
[b] Through a REST reconfiguration service over a loop-back TCP/IP connection.

Despite we could use the FSCRIPT execution engine's API to avoid the cost of sending the reconfiguration script, at least partially, having the REST reconfiguration service offers multiple possibilities for implementing different distributed reconfiguration schemes.

In the particular case of this application scenario, we judge the total settling-time of 157ms. as very acceptable given the goal of the QoS contract on readiness (i.e., contributing to the service availability). Interpreting the mean-time to reconfigure as the mean-time to recover (from failures), QOS-CARE represents considerable savings in time and cost in the automated management of software applications, especially when compared with the performance of a human performing the same task. In other words, this application scenario illustrates how QOS-CARE can be used to implement software systems with self-managed capabilities by satisfying their QoS contracts.

Moreover, even though the implied reconfigurations are very simple, the obtained dynamic capabilities constitute a reliable foundation for the Service Oriented Computing (SOC) and Service Oriented Architecture (SOA) paradigms.

7.3.6 Implementation Details

In this section we present some implementation details of the analyzed application example in terms of its components, classes, physical lines of code (LOC), and density (i.e., LOCs per class). As summarized in Table 7.5, this application example is composed of 5 SCA components, implemented with 3 Java interfaces and 22 Java classes, reusing the QOS-CARE monitor interface.

As previously explained, for this application example we reused two FRASCATI examples. To obtain the Web mashup application, we unified the interface for invoking the weather services from different weather information providers, and developed:

- One monitoring component (63 LOC), to notify QOS-CARE about a weather service unreadiness from the currently used weather information provider.

Table 7.5: Twitter-Weather mashup application example implementation details

Twitter-Weather Mashup Application Example	Components	Classes and Interfaces	LOC	LOC/ Class
Monitoring Elements	1	1	63	63
Application Logic[a]	4	24	1,512	63
Total	5	25	1,575	63

[a] Including only one weather information provider, and WSDL generated classes and interfaces.

- 3 application-logic components: one for orchestrating the weather service invocations among the different registered weather information providers (composite file of 87 total LOC); and two for implementing additional weather information providers, namely, for the Google and Yahoo ones (35 LOC average per composite file).

Therefore, apart from the QoS contract specification on the application readiness (including the 4 reconfiguration rules), the net development work for preserving the contract on the application example amounts to the monitoring component, that is, 63 LOC (i.e., 4% of the total), and the implementation of the additional weather information providers (58 LOC in average for each, 3.6% of the total). Given that all of our remaining work was devoted to augment the application logic and its own functionalities, this represents a very low amount of work compared to the total application size.

7.4 Analysis of QOS-CARE Limitations

In this section we present the limitations of QOS-CARE that we have identified from our own experience using it in the presented application scenarios.

Automated Interface Adaptation

As we have illustrated in both of the presented application scenarios, our mechanism for dynamic reconfiguration allows reconfiguration rules to dynamically deploy and un-deploy components with different services and functionalities. Nonetheless, although the reconfiguration rules can specify how to reconfigure the managed application to use these new services, the user still needs to develop or provide the actual implementation of the implied component to exploit the corresponding functionality. In some cases, such as the RVCS application, it is foreseeable the kind of components to be required by reconfiguration rules. However, in some others, such as the Internet mashup application, it is not possible to anticipate the particular interface specification that a service provider will use to implement a particular service. Thus, to leverage the QOS-CARE capabilities for dynamic reconfiguration it is necessary to have automated mechanisms to translate from implementations of one interface to another, or to standardize the interfaces for given services.

Unfortunately, this is a well known unsolved challenge shared with the research community on Component-Based Software Engineering.

Guaranteeing Additional Adaptation Properties

The main goal of this dissertation is the preservation of QoS contracts with reliable reconfiguration mechanisms. Given that our strategy to preserve QoS contracts is based on the application of design-patterns through dynamic reconfiguration, we defined the reliability in terms of five properties inherent to reconfiguration (i.e., adaptation) software systems. These properties are a subset of the adaptation properties that we identified and defined as part of this dissertation for this kind of systems.

Even though we identified, validated, and verified the conditions required to guarantee each of these five properties, it would be desirable to guarantee several other of the adaptation properties. However, according to the U.S. Air Force Science & Technology vision for the next future [Werner Dahm, 2010], developing mechanisms that guarantee these adaptation properties in general will take the research community on self-adaptive software systems and control engineering at least the next decade, if not more.

7.5 Chapter Summary

In this chapter we have presented two application scenarios for QOS-CARE: a reliable videoconference system (RVCS) and a dynamic Internet mashup application. We used these case studies to (i) validate the applicability and practical feasibility of our formal model and its realization as an SCA architecture and implementation, and (ii) show how do we complete the validation and verification of the conditions that characterize the adaptation properties through which we defined the reliability of the reconfiguration mechanism. Given the different application domains of the two application scenarios, being them complementary, we could evaluate different aspects of QOS-CARE with them.

Quantitative Results

The obtained MTTR results in both application scenarios are significant as indicative of the QOS-CARE's reliability, as measured quantitatively. On one hand, the reconfiguration settling-times are small, even when instrumented over an Internet connection, being them acceptable for the corresponding application domains. Nonetheless, depending on the scenario, several performance improvements can be obtained. For instance, it would be possible to send the reconfiguration scripts (and deploy new components) only on their first use, instead of sending them again (and removing them) each time, and just deactivate them for ulterior invocation or reuse. On the other hand, compared to the response times of existing Internet services, the overhead introduced by QOS-CARE while running the managed application in FRASCATIis negligible.

As a result, the experimental evaluation performed with these application scenarios confirm the applicability and practical feasibility of using QOS-CARE as a platform-independent SCA layer for preserving QoS contracts in component-based software applications.

Qualitative Results

As thoroughly explained, QOS-CARE enforces the separation of concerns between the managed application and the reconfiguration mechanism, and their corresponding properties. In both application scenarios we were able to:

- Specify the QoS contracts independently of the managed applications.
- Except for the monitor probes, the managed applications did not require of any modifications for the reconfiguration mechanism to operate on them in order to preserve their respective QoS contracts.
- The reconfiguration mechanism (i.e., QOS-CARE) remains explicitly separated from the managed application, and moreover, it is executed as a completely independent composite

in FRASCATI.

- Finally, this independence and separation of concerns allowed us to analyze the reliability adaptation properties of our reconfiguration mechanism. In particular, we successfully completed the validation and verification of the five adaptation properties with which we defined the reliability of QOS-CARE.

Part IV

Summary

Chapter 8

Conclusions and Future Work

Contents

In this concluding chapter we present an overview of this dissertation by summarizing the main challenges and goals addressed, and the corresponding achieved contributions. Finally, we discuss some further research opportunities opened by our research work.

8.1 Dissertation Summary

8.1.1 Addressed Challenges and Goals

Over the last years, software services have gained ubiquity and proliferated dramatically, giving rise to a new software industry promoted by the possibilities of their massive use and their pervasiveness in all aspects of everyday life. The proliferation and massive use of these services, individually or combined among them and with traditional ones (e.g., Web mashups and wrapped legacy applications), presents new challenges and requirements for the software engineering community. These new requirements, such as having to fulfill changing QoS levels under different context situations, further exacerbate the problem of guaranteeing the expected quality of these services at runtime. In effect, as introduced in Chapter 1, the most salient challenges for this new industry arise from the dependencies that these services' QoS properties have on the dynamic nature of context. In this dissertation we have addressed the following:

C1: The ***definition of adaptation properties*** (i.e., properties inherent to self-adaptive software (SAS)) as a common basis to evaluate, compare, improve, and even combine reconfiguration mechanisms. We also used these properties to guarantee the reliability of our reconfiguration mechanism.

C2: The ***reliable and autonomous preservation of QoS contracts*** through dynamic reconfiguration under varying conditions of system execution. As this corresponds to a problem of self-reconfiguration, this requires to define a reconfiguration mechanism that manages the dynamic evolution of both, context situations, and the software structures to reconfigure.

C3: The ***clear separation of concerns*** between the reconfiguration mechanism and the managed software application, as well as their corresponding properties, from architecture to implementation. Moreover, the elements of the MAPE-K model must also be explicit and visible in the reconfiguration mechanism.

C4: The ***management of context uncertainty***, as reconfiguration mechanisms should be robust with respect to (i) the unpredictability of context changes to be faced by the managed application; and (ii) the user-defined strategies (e.g., reconfiguration rules to be parameterized in the reconfiguration mechanism) to address the context changes to be faced by the managed system.

C5: The ***feasible and seamless integration*** of the implemented reconfiguration mechanism with existing component runtime platforms, for preserving QoS contracts in the executed component-based software applications.

In light of these challenges, we pursued two main goals: (i) characterize inherent properties to SAS systems; and (ii) provide a comprehensive solution for QoS contracts preservation through dynamic reconfiguration, built on a formal foundation. We refined these two main goals by adopting one specific goal for each of the challenges stated previously, as follows:

G1: Identify and define key properties inherent to software self-adaptation.

G2: Develop a formal model for the dynamic and reliable reconfiguration mechanism to preserve QoS contracts in component-based software.

G3: Maintain a clear separation of concerns between the reconfiguration mechanism and the managed software application, as well as between their corresponding properties.

G4: Guarantee robustness in the reconfiguration mechanism with respect to context unpredictability.

G5: Determine the practical feasibility and reusability of the formal based reconfiguration mechanism.

In the following section, we discuss our achieved contributions with respect to these goals and challenges.

8.1.2 Contributions

We classify the contributions achieved in this dissertation in two parts, according to our two main goals. In the first part, corresponding to the first challenge and main goal presented in the previous section, we characterized *inherent properties of self-adaptive software (SAS) systems*. Besides useful to evaluate adaptation mechanisms in standardized and comparable ways, these properties can be used to improve and combine these mechanisms. We characterized these properties from an extensive analysis of research papers and proposals of SAS systems, in which we identified that the engineering of this kind of systems has no standard properties to evaluate them. However, by its own nature, SAS adaptation mechanisms are essentially feedback loops as defined in control theory, and thus, they should be evaluated using the standard properties used to evaluate feedback loops, re-interpreted for the software domain.

In the second part, corresponding to the last four challenges and our second main goal, we have followed a comprehensive strategy to obtain QoS-CARE, a reliable and robust reconfiguration mechanism to preserve QoS contracts in component-based software applications. This strategy is composed of four aspects, focused in the planner element of the MAPE-K loop model: a *formal model*, its realization as a *software architecture*, the *implementation* of this architecture, and its corresponding *experimental evaluation*. By following this strategy, we effectively use *(formal) models at runtime* to reliably and robustly reconfigure software applications for preserving their QoS contracts. More specifically, we have shown the feasibility of exploiting *design patterns at runtime* in reconfiguration loops to fulfill expected QoS levels associated to specific context conditions. For this, we encode appropriate design patterns in left and right-hand sides of reconfiguration rules, given their determining influence on QoS properties and their levels of fulfillment. Finally, we realized our formal model as a component-based software architecture and its implementation, QoS-CARE, which can be used as an additional layer of SCA middleware stacks. We validated and verified the reliability of QoS-CARE in terms of five adaptation properties, and also evaluated it experimentally through its use in two application scenarios.

We detail the concrete achieved contributions as follows.

Formal Model. We conceived our formal model for QoS contracts preservation through dynamic reconfiguration by combining two formal systems: the theory of Finite State Machines (*FSM*) and the Typed Attributed Graph (called e-graphs) Transformation System (*TAGTS*) theory. On one hand, inspired by the MAPE-loop model, we extended the classic definition of FSMs to specify the semantics of QoS contracts in order to achieve reconfiguration autonomy in response to context changes that notify (imminent) contract violations. We manage *robustness with respect to context changes unpredictability* by modeling (i) not only the states of contract fulfillment, but also those of *unfulfillment*; and (ii) the reconfiguration of the managed application as state transitions. On the other hand, we modeled the reconfiguration operation as such by extending a TAGTS in order to exploit its capabilities of graph-based pattern-matching and transformation in the application of design patterns in our reconfiguration rules. We benefit from TAGTS formal properties to obtain reconfiguration *reliability*, as expressed in terms of the adaptation properties of termination, atomicity, and structural conformance. In this way, we complement the natural expressive power of TAGTS to specify reconfiguration rules for component-based software structures, with the one of FSMs for controlling context-driven and state-based systems.

By using this formal model to specify and develop our reconfiguration mechanism, we establish clear limits between it and the managed software application, and their corresponding properties. As a result, our reconfiguration mechanism is independent also from component runtime platforms, and reusable on any of them.

QoS-CARE Architecture, Implementation and Evaluation. We realized our formal model through a component-based software architecture and its respective implementation that creates and maintains an e-graph representation of the managed application and its corresponding QoS contract at runtime. More specifically, our implementation bridges the actual running application structure with the e-graph representation structure by using a pair of functions that maintain the coherence between these two structures. These functions are implementations of the corresponding specifications that result from the formal definitions given in Section 4.2.2. Moreover, as these functions allow the reconfiguration mechanism to be executed as decoupled from the component runtime platform, they determine the feasible and seamless integration of QoS-CARE with existing component platforms. In this sense, we conceived QoS-CARE as an SCA layer for dynamic reconfiguration to preserve QoS contracts in component-based software applications. In addition, the relevant elements of the MAPE-loop model remain explicit in its implementation.

To experimentally evaluate the performance of QoS-CARE, we configured it to be deployed and executed in FraSCAti, a multi-scale implementation of the SCA specification, to preserve QoS contracts in two application scenarios. Then, we designed and executed a benchmark to evaluate the mean-time to reconfigure (MTTR) and the overhead of QoS-CARE, by using this configuration on several use-cases of these application scenarios. From the obtained results, we confirmed the practical feasibility and (re)usability of our reconfiguration system, besides constituting an additional test for our formal model's soundness.

Adaptation Properties. In this part of our contribution, we developed a framework to classify and compare SAS systems. This framework defines a set of characterizing dimensions, a list of adaptation properties (mentioned previously), and respective mappings from these properties to quality attributes. We applied this framework in several representative research works on the engineering of self-adaptive software to refine the identified adaptation properties. The resulting properties, as inherent to SAS systems, provide a common basis to assess adaptation mechanisms, especially when this assessment has comparative, improvement, or even mix-and-match combination purposes. More importantly, in our opinion, this contribution constitutes a fundamental step towards the standardization of comparable assessment methods for SAS systems, based on well defined adaptation properties.

As applied to this dissertation, the most important properties that we characterized are the atomicity, the short settling-time, the termination, the structural consistency, and the robustness with respect to context unpredictability. We validated the reliability of our reconfiguration mechanism in terms of these properties.

8.2 Future Work

We close this dissertation with the identification and analysis of some further research opportunities opened by our research work.

8.2.1 Short-Term Opportunities

For the short-term, we consider the following opportunities:

Graphical and syntax-directed editors for reconfiguration rules. Graphs are widely recognized for their graphical notation to express software structures. We have used e-graphs (typed attributed graphs) for modeling component-based software structures and also the corresponding reconfiguration rules to exploit their graph visual presentations, graph-based pattern-matching and transformation capabilities. Despite the editors that we employed to specify the e-graph reconfiguration rules for the application scenarios are graphical and usable, it would be better to have an editor with a concrete syntax closer to a known components model (e.g., SCA) notation. This editor should be developed, based on an appropriate DSL, with automated tools to assist the user in the specification of reconfiguration rules. From the notation defined by this DSL, it could translate these rules into the e-graph notation.

Improved performance for mobile devices. For the reconfiguration settling-time, we obtained acceptable MTTR measurements for the two validation scenarios that we implemented as managed software applications, while verifying the preservation of the respective QoS contracts. However, it is worth noting that the reconfiguration settling-times were obtained in high computing-power devices. To be deployed in more modest devices, such as smart-phones, QoS-CARE should be optimized to improve significantly these measurements. This would imply, of course, to improve also the execution performance of the component runtime platforms to be used.

Scalability limits. A critical aspect for the adoption of our model is to evaluate its scalability. Although the results of our implementation's performance evaluation were acceptable, this evaluation was based on two relatively small application scenarios (6 components and 87 implementation classes in average), with simple yet plausible reconfiguration rules. Thus, to estimate the QoS-CARE scalability limits we would need to evaluate its performance (i.e., CPU and memory consumption) with larger application scenarios and more complex reconfiguration rules.

8.2.2 Long-Term Opportunities

As worth of future work in the long-term, we consider the following three aspects of our contribution.

Leveraging design-patterns at runtime. Since we have demonstrated the feasibility of exploiting design patterns at runtime to preserve QoS contracts using a structure-based strategy with QoS-CARE, we have opened the possibility of exploiting them at runtime also for other purposes. For instance, in contrast to our structure-based strategy but nonetheless exploiting it, design patterns at runtime could help to understand and develop quantitative behavior-based models of QoS properties on software systems. These formalized behavior models, as the system transfer function models built in control theory with self-tuning capabilities, could be used to predict characteristic responses of software systems to given context changes, such as a sudden large increment of service requests. This is a fundamental challenge to achieve stability and efficient resource use in self-adaptive software systems.

Improving robustness to context uncertainty. The states of contract *unfulfillment* managed by QoS-CARE constitutes only one step towards understanding how to achieve general robustness to context uncertainty and its related problems in self-adaptation. In addition, and related to these states, our characterization of *dependency relationships* among QoS contract conditions given in Section 4.3.2 should help to understand how to address the differences between *mutually independent* and *inter-dependent* contract conditions. Nonetheless, further analysis of context may reveal other relationships yet to be characterized for improving our robust preservation of QoS contracts.

Guaranteeing additional adaptation properties. One of the main goals of this dissertation is the preservation of QoS contracts with reliable reconfiguration mechanisms. Given that our strategy to preserve QoS contracts is based on the application of design patterns through dynamic reconfiguration, we defined the reliability in terms of five properties inherent to adaptive software systems. These properties are a subset of the adaptation properties that we identified and defined as part of this dissertation for this kind of systems. Even though we identified, validated, and verified the conditions required to guarantee these five properties for the two application scenarios, one of the most interesting opportunities for further research opened by this work is the development of a generalized formal framework to guarantee other of the adaptation properties that we characterized.

Bibliography

[Aho and Ullman, 1972] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. 86

[Andersson *et al.*, 2009] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 27–47. Springer-Verlag, 2009. 10, 45

[Appleby *et al.*, 2001] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Océano - sla based management of a computing utility. In *Procs. of 7th IFIP/IEEE Intl. Symp. on Integrated Network Management*, pages 855–868, 2001. 51, 52, 53, 54, 55, 56, 57, 60

[Avizienis *et al.*, 2004] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, January 2004. 17, 25, 111, 120, 128

[Bachmann *et al.*, 2000] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical Concepts of Component-Based Software Engineering. Volume 2. Technical Report CMU/SEI-2000-TR-008, CMU/SEI, 2000. 8, 25

[Barbacci *et al.*, 1995] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, CMU/SEI, 1995. 34, 41, 52, 53, 55, 56, 112

[Baresi and Guinea, 2011] Luciano Baresi and Sam Guinea. Self-Supervising BPEL Processes. *IEEE Trans. on Software Engineering*, 37:247–263, March 2011. 53, 54, 56, 57, 60

[Bass *et al.*, 2003] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, Mass, second edition, 2003. 12, 66

[Becker, 2008] Steffen Becker. Quality of Service Modeling Language. In Irene Eusgeld, Felix Freiling, and Ralf Reussner, editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 43–47. Springer Berlin / Heidelberg, 2008. 31

[Beisiegel *et al.*, 2007a] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, et al. Service component architecture, assembly model specification. Specification Version 1.0, Open Service Oriented Architecture Collaboration, 2007. 8, 12, 17, 26, 164

[Beisiegel *et al.*, 2007b] Michael Beisiegel, Dave Booz, Ching-Yun Chao, Mike Edwards, et al. Sca policy framework. Specification Version 1.0, Open Service Oriented Architecture Collaboration, 2007. 27, 30

[Bell, 2008] Michael Bell. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*. Wiley Publishing, 2008. 8, 164

[Bentancour *et al.*, 2011] Martin Bentancour, Libor Cada, Jing Wen Cui, Marcio d'Amico, Ural Emekci, Sebastian Kapciak, Jennifer Ricciuti, and Margaret Ticknor. WebSphere Application Server V8: Administration and Configuration Guide. Technical report, IBM Corporation, 2011. 30

[Beugnard *et al.*, 1999] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999. 7, 30, 31, 71, 163

[Bosch, 2000] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 30

[Braga *et al.*, 2009] Christiano Braga, Fabricio Chalub, and Alexandre Sztajnberg. A formal semantics for a quality of service contract language. *Electronic Notes of Theoretical Computer Science*, 203(7):103–120, 2009. 31, 33

[Bruggink, 2008] H.J. Sander Bruggink. Towards a Systematic Method for Proving Termination of Graph Transformation Systems. *Electronic Notes Theoretical Computer Science*, 213:23–38, May 2008. 115

[Bruneton *et al.*, 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java: Experiences with Auto-Adaptive and Reconfigurable Systems. *Software Practice & Experience*, 36(11-12):1257–1284, 2006. 26, 28, 39

[Bucchiarone *et al.*, 2009] Antonio Bucchiarone, Patrizio Pelliccione, Charlie Vattani, and Olga Runge. Self-repairing systems modeling and verification using agg. In *IEEE/IFIP WICSA/ECSA*, pages 181–190. IEEE, 2009. 33, 115

[Buschmann *et al.*, 2007] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*. John Wiley & Sons, 2007. 12, 34, 66, 74, 131, 169

[Candea *et al.*, 2004] George Candea, James Cutler, and Armando Fox. Improving Availability with Recursive Microreboots: A Soft-State System Case Study. *Performance Evaluation Journal*, 56(1-3):213 – 248, 2004. 17, 25, 39, 51, 53, 55, 56, 57, 60, 112

[Cansado *et al.*, 2010] Antonio Cansado, Carlos Canal, Gwen Salaün, and Javier Cubo. 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. 31, 33

[Caprarescu and Petcu, 2009] Bogdan Alexandru Caprarescu and Dana Petcu. A self-organizing feedback loop for autonomic computing. *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World*, 0:126–131, 2009. 38

[Cardellini *et al.*, 2009] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. QoS-Driven Runtime Adaptation of Service Oriented Architectures. In *Procs. of 7th Joint Meeting European Software Engineering Conf. and ACM Symp. on Foundations of Software Engineering*, ESEC/FSE '09, pages 131–140. ACM, 2009. 39, 51, 53, 54, 56, 57, 60

[Chang and Collet, 2007a] Hervé Chang and Philippe Collet. Compositional patterns of non-functional properties for contract negotiation. *JSW*, 2(2):52–63, 2007. 33

[Chang and Collet, 2007b] Hervé Chang and Philippe Collet. Patterns for integrating and exploiting some non-functional properties in hierarchical software components. In *Procs. of 14th IEEE Intl. Conference and Workshops on the ECBS'07*, pages 83–92. IEEE Computer Society, 2007. 31

[Chang *et al.*, 2006] Hervé Chang, Philippe Collet, Alain Ozanne, and Nicolas Rivierre. From components to autonomic elements using negotiable contracts. In *3rd Intl. Conf. ATC*, volume 4158 of *LNCS*, pages 78–89, 2006. 8, 33, 164

[Cheng *et al.*, 2009a] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, LNCS, pages 1–26. Springer-Verlag, 2009. 10, 24, 34, 35, 37, 39, 40, 61, 68, 69

[Cheng *et al.*, 2009b] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Procs. of 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE Computer Society, 2009. 10, 39, 45

[Clements and Shaw, 2009] Paul Clements and Mary Shaw. The Golden Age of Software Architecture Revisited. *IEEE Softw.*, 26:70–72, July 2009. 12, 66

[Collet *et al.*, 2005] Philippe Collet, Roger Rousseau, Thierry Coupaye, and Nicolas Rivierre. A Contracting System for Hierarchical Components. In *Procs. of 8th Intl. Symp. of Component-Based Software Engineering*, volume 3489 of *LNCS*, pages 187–202. Springer, 2005. 7, 10, 25, 31, 34, 71, 163

[Collet *et al.*, 2007] Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite contract enforcement in hierarchical component systems. In *6th Intl. Symp. on Software Composition (SC)*, volume 4829 of *LNCS*, pages 18–33, 2007. 8, 164

[Colombo *et al.*, 2006] Massimiliano Colombo, Elisabetta Di Nitto, and Marco Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In Asit Dan and Winfried Lamersdorf, editors, *Intl. Conf. on Service-Oriented Computing*, volume 4294 of *LNCS*, pages 191–202. Springer, 2006. 81

[Comuzzi and Pernici, 2009] Marco Comuzzi and Barbara Pernici. A framework for qos-based web service contracting. *ACM Trans. on the Web*, 3(3):1–52, 2009. 31

[de Lemos *et al.*, 2012] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Bradley Schmerl, Dennis B. Smith, João P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*. Springer, 2012. 10, 13, 18, 24, 35, 37, 39, 40, 61

[Delaval and Rutten, 2010] Gwenaël Delaval and Éric 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, pages 93–112, 2010. 10, 32, 34, 39

[Dougherty *et al.*, 2009] Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, and Kazuya Togashi. Secure Design Patterns. Technical Report CMU/SEI-2009-TR-010, CMU/-SEI, CERT Program, 2009. 12, 66, 74, 122, 169

[Dowling and Cahill, 2004] Jim Dowling and Vinny Cahill. Self-managed decentralised systems using k-components and collaborative reinforcement learning. In *Procs. of 1st ACM SIGSOFT Workshop on Self-Managed Systems*, WOSS '04, pages 39–43, New York, NY, USA, 2004. ACM. 10, 51, 52, 53, 55, 56, 57, 60

[Dumont and Huzmezan, 2002] G.A. Dumont and M. Huzmezan. Concepts, methods and techniques in adaptive control. In *The 2002 American Control Conference*, volume 2, pages 1137 – 1150. IEEE, 2002. 49

[Ehrig *et al.*, 2004] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In *Proc. of ICGT'04*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004. 114

[Ehrig *et al.*, 2005] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination Criteria for Model Transformation. In *Procs. of 8th Intl. Conf. Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 49–63. Springer, 2005. 115

[Ehrig *et al.*, 2009] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2009. 15, 63, 68, 114, 115, 167

[Ehrig *et al.*, 2010] Hartmut Ehrig, Claudia Ermel, Olga Runge, Antonio Bucchiarone, and Patrizio Pelliccione. Formal analysis and verification of self-healing systems. In *Procs. of 13th Intl. Conf. Fundamental Approaches to Software Engineering*, volume 6013 of *LNCS*, pages 139–153. Springer, 2010. 33, 52, 53, 55, 57, 60

[Fiadeiro and Lopes, 2010] José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 70–85, Berlin, Heidelberg, 2010. Springer-Verlag. 33

[Filieri *et al.*, 2010] Antonio Filieri, Carlo Ghezzi, Vincenzo Grassi, and Raffaela Mirandola. Reliability Analysis of Component-Based Systems with Multiple Failure Modes. In *Procs. of 13th*

Intl. Symp. Component-Based Software Engineering, volume 6092, pages 1–20. Springer, 2010. 25, 112

[Floch *et al.*, 2006] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23:62–70, March 2006. 51, 52, 53, 55, 57, 60

[Floyd, 1967] Robert Floyd. Assigning Meaning to Programs. In *Proc. of Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967. 32

[Frølund and Koistinen, 1998] Svend Frølund and Jari Koistinen. Quality of services specification in distributed object systems design. In *Procs. of 4th Conf. on Object-Oriented Technologies and Systems*, volume 4, pages 179–202. USENIX Association, 1998. 31

[Garlan *et al.*, 2003] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. *Lecture Notes in Computer Science*, 2677, 2003. 38

[Garlan *et al.*, 2004] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004. 39, 53, 57, 60

[Gat, 1998] Erann Gat. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, pages 1–11. MIT Press, 1998. 38

[Giese *et al.*, 2009] H. Giese, Y. Brun, J. Di Marzo Serugendo, C. Gacek, H.M. Kienle, H.A. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive and self-managing systems. *LNCS 5527, Springer-Verlag*, pages 47–69, 2009. 37

[Goldsby and Cheng, 2008] Heather J. Goldsby and Betty H. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 568–583. Springer-Verlag, 2008. 34, 39

[González, 2011] Oscar González. *Monitoring and Analysis of Workflow Applications: A Domain-Specific Language Approach*. PhD thesis, Universidad de los Andes and Vrije Universiteit Brussel, 2011. 99

[Grassi *et al.*, 2009] Vincenzo Grassi, Raffaela Mirandola, and Enrico Randazzo. Model-Driven Assessment of QoS-Aware Self-Adaptation. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 201–222. Springer-Verlag, 2009. 10, 45

[Harel, 1987] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987. 81

[Harel, 1988] David Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, May 1988. 87

[Heckel *et al.*, 2002] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proceedings of the First International Conference on Graph Transformation*, ICGT '02, pages 161–176. Springer-Verlag, 2002. 114

[Heineman and Councill, 2001] George T. Heineman and William T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman, 2001. 8, 10, 24, 25, 34, 61, 164

[Hellerstein *et al.*, 2004] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. IEEE Press, John Wiley & Sons, 2004. 13, 17, 24, 25, 35, 36, 47, 49, 50, 51, 164

[Hellerstein *et al.*, 2009] Joseph L. Hellerstein, Sharad Singhal, and Qian Wang. Research Challenges in Control Engineering of Computing Systems. *IEEE Transactions on Network and Service Management*, 6(4):206–211, 2009. 47

[Hermosillo *et al.*, 2010] Gabriel Hermosillo, Lionel Seinturier, and Laurence Duchien. Using Complex Event Processing for Dynamic Business Process Adaptation. In *Procs. of 7th Intl. Conf. on Services Computing*, SCC, pages 466–473. IEEE CS, July 2010. 36, 100

[Hnětynka and Plášil, 2006] Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In *Procs. of 2006 Intl. Symposium on Component-Based Software Engineering*, volume 4063 of *LNCS*, pages 352–359. Springer-Verlag, 2006. 32

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 32, 62

[Hopcroft *et al.*, 2006] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 15, 63, 76, 167

[Huang *et al.*, 2011] Gang Huang, Weihu Wang, Tiancheng Liu, and Hong Mei. Simulation-based Analysis of Middleware Service Impact on System Reliability: Experiment on Java Application Server. *Journal of Systems and Software*, pages 1–11, 2011. 25, 112

[IBM Corporation, 2006] IBM Corporation. An Architectural Blueprint for Autonomic Computing. Technical report, IBM Corporation, June 2006. 24, 35, 37, 38

[Inverardi *et al.*, 2009] Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Towards an assume-guarantee theory for adaptable systems. In *Procs. of 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 106–115. IEEE Computer Society, 2009. 12

[ISO, 2001] The ISO. ISO/IEC 9126-1:2001 Software Engineering - Product Quality - Part 1: Quality Model. Standard, International Organization for Standardization (ISO), 2001. 31, 112

[Jacklin *et al.*, 2004] Stephen A. Jacklin, Michael R. Lowry, Johann M. Schumann, Pramod P. Gupta, John T. Bosworth, Eddie Zavala, and John W. Kelly. Verification, Validation, and Certification Challenges for Adaptive Flight-Critical Control System Software. In *Procs. of American Institute of Aeronautics and Astronautics AIAA Guidance Navigation and Control Conference and Exhibit*. American Institute of Aeronautics and Astronautics, 2004. 13, 164

[Jureta *et al.*, 2009] Ivan J. Jureta, Caroline Herssens, and Stéphane Faulkner. A comprehensive quality model for service-oriented systems. *Software Quality Control*, 17:65–98, March 2009. 25, 32

[Kaddoum *et al.*, 2010] Elsy Kaddoum, Claudia Raibulet, Jean-Pierre Georgé, Gauthier Picard, and Marie-Pierre Gleizes. Criteria for the evaluation of self-* systems. In *Procs. of 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'10, pages 29–38. ACM, 2010. 10, 45

[Keller and Ludwig, 2003] Alexander Keller and Heiko Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Networking and Systems Management*, 11(1):57–81, 2003. 7, 31, 71, 163

[Kephart and Chess, 2003] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003. 7, 8, 15, 24, 35, 72, 163, 164

[Kircher and Jain, 2004] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, 2004. 12, 66

[Krakowiak, 2009] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*. http://sardes.inrialpes.fr/~krakowia/MW-Book, 2009. 7, 12, 31, 66, 71, 74, 103, 163, 169

[Kramer and Magee, 2007] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268. IEEE CS, 2007. 7, 38, 39, 163

[Kramer and Magee, 2009] Jeff Kramer and Jeff Magee. A rigorous architectural approach to adaptive software engineering. *J. Computer Science Technology*, 24(2):183–188, 2009. 73

[Kruchten *et al.*, 2006] Philippe Kruchten, Henk Obbink, and Judith Stafford. The Past, Present, and Future for Software Architecture. *IEEE Software*, 23:22–30, March 2006. 12, 66

[Kumar *et al.*, 2007] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. *Cluster Computing*, 10:443–455, December 2007. 51, 53, 56, 57, 60

[Lee *et al.*, 2009] Jae Yoo Lee, Jung Woo Lee, Du Wan Cheun, and Soo Dong Kim. A Quality Model for Evaluating Software-as-a-Service in Cloud Computing. In *Procs. of 7th ACIS Intl. Conf. on Software Engineering Research, Management and Applications*, SERA '09, pages 261–266. IEEE Computer Society, 2009. 31

[Léger *et al.*, 2010] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *Procs. of 13th Intl. Symp. of Component-Based Software Engineering*, volume 6092 of *LNCS*, pages 74–92. Springer, 2010. 10, 17, 34, 39, 52, 53, 57, 60

[Lin *et al.*, 2009] Xiangtao Lin, Bo Cheng, and Junliang Chen. A situation-aware approach for dealing with uncertain context-aware paradigm. In *Procs. of 28th IEEE Conf. on Global Telecommunications*, GLOBECOM'09, pages 1880–1885. IEEE Press, 2009. 40

[Litoiu *et al.*, 2005] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical Model-Based Autonomic Control of Software Systems. In *Procs. of 2005 Workshop on Design and Evolution of Autonomic Application Software*, DEAS '05, pages 1–7. ACM, 2005. 38

[Lu *et al.*, 2000] Chenyang Lu, John A. Stankovic, Tarek F. Abdelzaher, Gang Tao, Sang H. Son, and Michael Marley. Performance specifications and metrics for adaptive real-time systems. In *Real-Time Systems Symposium*, 2000. 51, 56

[Luckham, 2001] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2001. 36, 100

[Ludwig *et al.*, 2003] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web Service Level Agreement (WSLA) Language Specification, 2003. IBM Available Specification. 31

[McKinley *et al.*, 2004] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004. 8, 24, 164

[Meng, 2000] Alex C. Meng. On evaluating self-adaptive software. In *Procs. of 1st Intl. Workshop on Self-Adaptive Software*, IWSAS' 2000, pages 65–74. Springer-Verlag New York, Inc., 2000. 10, 13, 51, 164

[Merle *et al.*, 2011] Philippe Merle, Romain Rouvoy, and Lionel Seinturier. A Reflective Platform for Highly Adaptive Multi-Cloud Systems. In *10th Intl. Workshop on Adaptive and Reflective Middleware*, ARM'2011 at the 12th ACM/IFIP/USENIX Intl. Middleware Conference, pages 1–7, 2011. 27

[Meyer, 1992] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. 32

[Mukhija and Glinz, 2005] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of 18th International Conference on Architecture of Computing Systems*, 2005. 52, 53, 57, 60

[Müller *et al.*, 2008] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of Control in Adaptive Systems. In *Procs. of 2nd Intl. Workshop on Ultra-Large-Scale Software-Intensive Systems*, ULSSIS'08, pages 23–26. ACM, 2008. 10, 37, 38

[Müller *et al.*, 2009] Hausi A. Müller, Holger M. Kienle, and Ulrike Stege. Autonomic Computing: Now you see it, now you don't—Design and evolution of autonomic software systems. *LNCS*, 5413:32–54, 2009. 38, 40

[Murray *et al.*, 2003] Richard M. Murray, Karl J. Åström, Stephen P. Boyd, Roger W. Brockett, and Gunter Stein. Future Directions in Control in an Information Rich World. *IEEE Control Systems*, 23:20–33, 2003. 34, 39

[Narendra and Balakrishnan, 1997] Kumpati S. Narendra and Jeyendran Balakrishnan. Adaptive control using multiple models. *IEEE Transactions on Automatic Control*, 42:171–187, 1997. 49

[Ogata, 1996] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, 3 edition, 1996. 25, 34, 36

[Oreizy *et al.*, 1999] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. 34

[Papazoglou *et al.*, 2007] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40:38–45, November 2007. 7, 27, 30, 163

[Parekh *et al.*, 2002] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Syst.*, 23:127–141, July 2002. 51, 53, 54, 55, 56, 57, 59, 60

[Ramachandran, 2002] Jay Ramachandran. *Designing Security Architecture Solutions*. John Wiley & Sons, Inc., 2002. 12, 41, 66, 74, 122, 169

[Reinecke *et al.*, 2010] Philipp Reinecke, Katinka Wolter, and Aad van Moorsel. Evaluating the adaptivity of computing systems. *Performance Evaluation*, 67(8):676 – 693, 2010. Special Issue on Software and Performance. 56

[Reussner *et al.*, 2003] Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reliability Prediction for Component-Based Software Architectures. *Journal of Systems and Software*, 66:241–252, June 2003. 25, 112

[Romero, 2011] Daniel Romero. *Context as a Resource: A Service-Oriented Approach for Context-Awareness*. Phd thesis, University of Science and Technology of Lille, July 2011. 30

[Röttger and Zschaler, 2003] Simone Röttger and Steffen Zschaler. CQML+: Enhancements to CQML. In *Procs. of 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering*, pages 43–56. Cépaduès-Éditions, 2003. 31

[Salehie and Tahvildari, 2009] Mazeiar Salehie and Ladan Tahvildari. Self-aDaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4:14:1–14:42, May 2009. 7, 10, 37, 39, 45, 52, 112, 163

[Seinturier *et al.*, 2009] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Procs. of 6th Intl. Conf. on Services Computing*, SCC'09, pages 268–275. IEEE, 2009. 12, 16, 28, 30, 67, 99, 171

[Seinturier *et al.*, 2012] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience (SPE)*, pages 1–26, 2012. 24, 28, 29, 30

[Shaw and Garlan, 1996] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996. 12, 28, 66

[Shaw, 1994] Mary Shaw. Beyond objects: A software design paradigm based on process control. Technical report, Carnegie Mellon University, 1994. 34, 37, 38

[Sicard *et al.*, 2008] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using Components for Architecture-Based Management: the Self-Repair Case. In *Procs. of 30th Intl. Conf. on Software Engineering*, ICSE'08, pages 101–110. ACM, 2008. 39, 53, 57, 59, 60

[Smith, 1984] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Procs. of 11th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, POPL '84, pages 23–35. ACM, 1984. 28

[Solomon *et al.*, 2007] Bogdan Solomon, Dan Ionescu, Marin Litoiu, and Mircea Mihaescu. A real-time adaptive control of autonomic computing environments. In *Procs. of 17th Annual Intl. Conf. of the Centre for Advanced Studies Research*, CASCON 2007, pages 124–136, 2007. 38

[Solomon *et al.*, 2010] Andrei Solomon, Marin Litoiu, Jay Benayon, and Alex Lau. Business process adaptation on a tracked simulation model. In *Procs. of 2010 Conf. of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 184–198. ACM, 2010. 51, 53, 54, 57, 60

[Szyperski, 1998] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, 1998. 8, 10, 24, 25, 34, 61, 164

[Taentzer, 2004] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Procs. of 2003 Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 446–453. Springer-Verlag, 2004. 68, 101, 115

[Tamura and Cleve, 2010] Gabriel Tamura and Anthony Cleve. A Comparison of Taxonomies for Model Transformation Languages. *Paradigma – Revista Electrónica en Construcción de Software*, 4(1):1–14, 2010. 18

[Tamura *et al.*, 2011a] G. Tamura, R. Casallas, A. Cleve, and L. Duchien. QoS Contract-Aware Reconfiguration of Component Architectures Using E-Graphs. In *Procs. of 7th Intl. Workshop on Formal Aspects of Component Software*, volume 6921 of *LNCS*, pages 34–52. Springer, 2011. 13, 18, 34

[Tamura *et al.*, 2011b] Gabriel Tamura, Rubby Casallas, Anthony Cleve, and Laurence Duchien. Realizing QoS Contracts-Preservation through Dynamic Reconfiguration Based on Formal Models. *Journal Science of Computer Programming (SCP)*, pages 1–30, 2011. Article In Evaluation. 13, 18

[Tamura *et al.*, 2012] Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, João P. Sousa, Basil Becker, Mauro Pezzè, Gabor Karsai, Serge Mankovskii, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*. Springer, 2012. 13, 18

[Taylor *et al.*, 2009] Richard N. Taylor, Nenad Medvidovic, and Peyman Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA'09*, pages 171–180. IEEE, 2009. 34

[Tran and Tsuji, 2009] Vuong Xuan Tran and Hidekazu Tsuji. A Survey and Analysis on Semantics in QoS for Web Services. In *Intl. Conf. on Advanced Information Networking and Apps.*, pages 379–385. IEEE, 2009. 7, 25, 31, 71, 163

[Villegas *et al.*, 2011a] Norha Villegas, Hausi Müller, and Gabriel Tamura. Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring. In *Procs. of IEEE Intl. Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011)*, pages 1–10. IEEE, 2011. 99

[Villegas *et al.*, 2011b] Norha Villegas, Hausi Müller, Gabriel Tamura, Laurence Duchien, and Rubby 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*, SEAMS, pages 80–89. ACM, 2011. 10, 13, 18, 38, 39, 45, 111

[Villegas *et al.*, 2012] Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*. Springer, 2012. 13, 18, 24, 35, 73

[Vromant *et al.*, 2011] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On Interacting Control Loops in Self-Adaptive Systems. In *Procs. of 6th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, (SEAMS 2011), pages 202–207. ACM, 2011. 38

[Werner Dahm, 2010] Werner Dahm. Technology Horizons: a Vision for Air Force Science & Technology During 2010-2030. Technical report, U.S. Air Force, May 2010. 8, 39, 47, 138, 164

[Weyns *et al.*, 2010] Danny Weyns, Sam Malek, and Jesper Andersson. On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future. In *Procs. of 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2010, pages 84–93. ACM, 2010. 38

[White, 2005] Jules White. Simplifying autonomic enterprise java bean applications via model-driven development: a case study. In *The Journal of Software and System Modeling*, pages 601–615, 2005. 53, 55, 56, 57, 60

[Yacoub *et al.*, 2004] S. Yacoub, B. Cukic, and H.H. Ammar. A Scenario-based Reliability Analysis Approach for Component-based Software. *IEEE Transactions on Reliability*, 53(4):465 – 480, dec. 2004. 25, 112

[Yang *et al.*, 2009] Jie Yang, Gang Huang, Wenhui Zhu, Xiaofeng Cui, and Hong Mei. Quality attribute tradeoff through adaptive architectures at runtime. *Journal of Systems and Software*, 82:319–332, February 2009. 34, 100

[Zeng *et al.*, 2004a] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineeering*, 30:311–327, May 2004. 10, 34

[Zeng *et al.*, 2004b] Wenjun Zeng, Xinhua Zhuang, and Junqiang Lan. Network Friendly Media Security: Rationales, Solutions, and Open Issues. In *Procs. of 2004 Intl. Conf. on Image Processing (ICIP)*, pages 565–568. IEEE, 2004. 12, 41, 66

[Zschaler, 2004] Steffen Zschaler. Semantic concepts for the specification of non-functional properties of component-based software. In *Procs. of 26th Intl. Conf. on Software Engineering*, pages 51–53. IEEE Computer Society, 2004. 8, 164

Part V

Appendices

Appendix A

QoS-CARE: Un Système Fiable pour la Préservation de Contrats de Qualité de Service

A.1 Introduction

Au cours des dernières années, les services logiciels ont envahi de plus en plus tous les aspects de la vie quotidienne. Les possibilités croissantes de l'informatique omniprésente exigent des capacités très dynamiques de ces services logiciels pour satisfaire les exigences dépendantes du contexte des diverses conditions d'exécution du système. En outre, il est nécessaire que ces capacités dynamiques puissent être fiables et effectuées opportunément au moment même de l'exécution pour être acceptable pour les utilisateurs. Des exemples de ces capacités sont disponibles dans les entreprises qui suivent le paradigme d'Informatique Orientée par Service (ou Service Orientes computing en anglais et SOC par son acronyme) [Papazoglou *et al.*, 2007], pour lequel des services basés sur les composants sont continuellement découverts, (re)composés et consommés au moment de l'exécution. La recomposition des services de composants varie en accord avec les changements des conditions de contexte comme ceux des points d'accès au réseau, la localisation des utilisateurs, la date et même les intérêts associés à des endroits donnés (par exemple, en train d'acheter des souvenirs à l'aéroport le jour même du voyage de retour).

Les problèmes sous-entendus ci-dessus sur la fiabilité et les capacités dynamiques pour les services à base de composants nécessitent la prise en compte de deux facteurs clés. D'une part, les contrats de qualité de service (QoS) sont un moyen naturel et efficace pour la capture de ce type de conditions dépendantes du contexte [Beugnard *et al.*, 1999, Keller and Ludwig, 2003, Collet *et al.*, 2005, Krakowiak, 2009, Tran and Tsuji, 2009]. D'autre part, pendant la dernière décennie, l'ingénierie du logiciel auto-adaptatif (Self-Adaptive Systems ou SAS par son acronyme en anglais) a démontré des avances considérables pour la gestion des capacités dynamiques des supports d'exécution de façon à satisfaire les buts dépendants du contexte, comme ceux de l'auto-configuration, auto-guérison, auto-protection, et auto-optimisation [Kephart and Chess, 2003, Kramer and Magee, 2007, Salehie and Tahvildari, 2009]. En effet, ces deux facteurs clés constituent les piliers fondamentaux de la promesse des approches à base de composant pour le développement et l'évolution de logiciels. Cependant, dans le génie logiciel à base de composants (Component-Based Software Engineering ou CBSE par son acronyme en anglais) —aussi appelé développement à base de composants (Component-Based Development ou CBD par son acronyme en anglais) [Szyperski, 1998, Heineman and Councill, 2001], et les architectures orientées services (Service Oriented Architecture ou SCA pour son acronyme en anglais) [Beisiegel *et al.*, 2007a, Bell, 2008], ces deux facteurs ont été considérés avec différents objectifs et de façon différente [McKinley *et al.*, 2004, Zschaler, 2004, Chang *et al.*, 2006, Collet *et al.*, 2007]. Néanmoins, le paradigme de Composant a besoin de théories sur l'auto-adaptation qui soient conscientes du contrat QoS, globales et solides, et de modèles et mécanismes fiables, extensibles et réutilisables afin d'accomplir leur promesse.

Garantir la *satisfaction continuelle* de contrats fonctionnels et extra-fonctionnels dans des conditions d'exécution du système variable est l'un des principaux propos de l'ingénierie de systèmes logiciels auto-adaptatifs. Ce propos est mené par le modèle de monitoring-analyse-planning-exécution et connaissance partagée (MAPE-K ou tout simplement MAPE par son acronyme en anglais), qui a été adopté comme un concept central inspiré par les principes de la théorie de contrôle [Kephart and Chess, 2003]. Cependant, pour avancer dans la réalisation de cette vision, l'auto-adaptation a encore besoin de résoudre des défis clés tels que (i) les limitations venant des plans d'adaptation produits manuellement de façon à répondre à l'évolution dynamique et parallèle des situations de contexte et des structures des logiciels ; (ii) les limites souvent floues entre les mécanismes d'adaptation et les applications gérées qui gènent l'analyse de leurs propriétés respectives ; (iii) le contrôle limité de l'incertitude pour répondre à des changements de contexte imprévus ; et (iv) le manque de propriétés standards qui limitent l'évaluation comparable de mécanismes d'adaptation et leur effectivité pour réussir les buts d'adaptation [Werner Dahm, 2010].

Compte-tenu de ces défis, la contribution de cette thèse est double. Premièrement, nous fournissons une stratégie de solution globale pour la préservation des contrats QoS dans les applications de logiciels basés sur les composants, en reconfigurant dynamiquement leur architecture, c'est-à-dire en déployant/repliant les composants et en connectant/déconnectant les interfaces de service. En deuxième lieu, nous identifions et nous proposons des propriétés inhérentes aux logiciels auto-adaptatifs, c'est-à-dire des propriétés d'adaptation qui peuvent être utilisées pour standardiser des modèles d'évaluation comparables à ces types de système. Notre stratégie de solution globale comprend quatre éléments, concentrés sur l'élément du modèle de planning en boucle MAPE-K : (i) un *modèle formel*, (ii) sa réalisation sous forme d'une *architecture logicielle*, (iii) *l'implantation* de l'architecture logicielle à l'aide d'une couche SCA pour gérer la reconfiguration dynamique de façon à préserver les contrats QoS, et (iv) une *validation théorique et expérimentale* de cette solution.

Dans les sections suivantes, nous détaillons ces deux contributions, en commençant par les propriétés d'adaptation et, finalement, en définissant notre solution que nous avons appelée QoS-CARE : un Système fiable pour la préservation de contrats de QoS (ou QoS Contract-Aware Reconfiguration systEm par son acronyme en anglais).

A.2 Propriétés de Logiciels Auto-Adaptatifs

Dans cette partie, nous discutons du manque de propriétés standarisées inhérentes aux logiciels auto-adaptatifs. Ces propriétés pourraient être employées pour l'évaluation comparée de ces logiciels et pour l'amélioration de la composition de différents mécanismes d'adaptation. Pour cela, nous commençons par considérer les propriétés définies par la théorie de contrôle [Meng, 2000, Hellerstein *et al.*, 2004, Jacklin *et al.*, 2004], en les réinterprétant pour les logiciels auto-adaptatifs. Puis, nous les complétons ces propriétés par d'autres travaux de recherche représentatifs dans la communauté SAS. La contribution de cette section est un cadre d'évaluation des approches SAS, qui contient (i) un ensemble de dimensions de caractérisation, et (ii) une liste de définitions consolidées pour les propriétés d'adaptation identifiées.

A.2.1 Dimensions de Caractérisation pour les Logiciels Auto-Adaptatifs

Notre cadre d'évaluation pour les systèmes de logiciels auto-adaptatifs (SAS) possède cinq dimensions de caractérisation qui résultent de l'analyse du diagramme de blocs de la boucle de rétroaction représentée dans la Fig A.1. Pour chacune de ces dimensions, nous avons identifié une liste d'options de classement à partir d'attributs utilisés dans la théorie de contrôle, mais également d'autres proposés par des sources reconnues (par exemple, le Software Engineering Institute –SEI par son acronyme en anglais) et de l'analyse de plus de 80 articles de recherche[37].

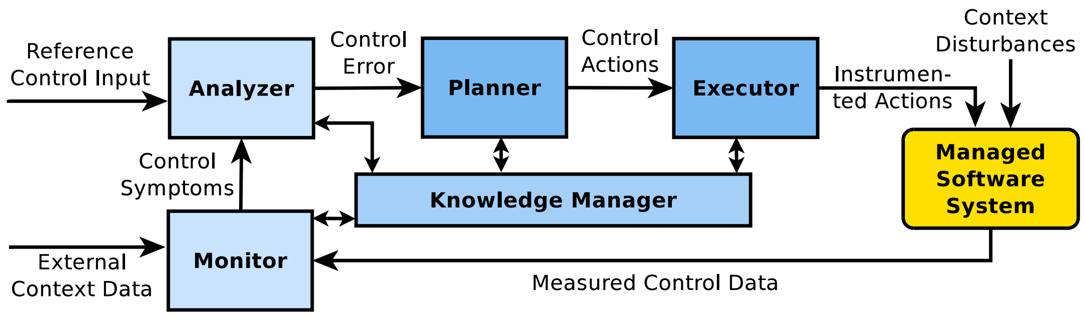


Figure A.1: Les dimensions de caractérisation pour les systèmes SAS identifiés dans le diagramme de boucle de rétroaction.

Buts d'adaptation. Même si ce n'est pas explicite dans la figure, les buts sont les raisons principales pour que le système ou l'approche soit auto-adaptative, tels que les auto-* buts (par exemple, auto-configuration, auto-guérison, auto-optimisation et auto-contrôle), ou la préservation des propriétés de qualité de service (QoS).

Entrées de référence de contrôle. L'ensemble concret et spécifique de valeurs et types correspondants employés pour spécifier l'état à atteindre et pour le maintenir dans l'application de contrôle par le mécanisme d'adaptation, soumis à des conditions variables d'exécution du système. Les entrées de référence peuvent être spécifiées, par exemple, comme des valeurs de référence simples, ou bien comme des accords de niveau de service (SLA).

Information de contrôle mesurée. L'ensemble de valeurs et types correspondants qui sont mesurés dans l'application gérée, tels que les domaines continus pour variables ou signaux simples, ou bien des expressions logiques ou des conditions pour les états des contrats.

Actions de contrôle. Celles-ci se caractérisent par la nature des actions requises pour contrôler ou modifier l'application gérée, en déterminant la sortie du planning d'adaptation. Cette sortie est instrumentée dans l'application gérée pour avoir un effet attendu sur elle-même. Les actions de contrôle peuvent être encodées à la main au moment de la conception ou synthétisées dynamiquement au moment de l'exécution, et peuvent être, par exemple, des signaux continus qui affectent le comportement de l'application gérée ou bien des opérations discrètes qui affectent l'infrastructure ordinatrice qui exécute l'application gérée.

[37]La description complète des dimensions de caractérisation, avec les options de classement peuvent se trouver sur la Section 3.2 de ce document.

Structure de système. Les systèmes auto-adaptatifs possèdent deux sous-systèmes bien définis (même s'ils sont possiblement impossibles à distinguer l'un de l'autre au niveau de l'implémentation) : (i) le mécanisme d'adaptation, et (ii) l'application gérée. Les approches analysées peuvent se regrouper en deux catégories : celles qui modélisent la structure de l'application gérée pour influencer son comportement en modifiant sa structure, et celles qui modélisent le comportement de l'application gérée pour l'influencer directement.

A.2.2 Définitions Consolidées pour les Propriétés d'Adaptation Identifiées

Nous classons les propriétés d'adaptation selon *comment* et *où* elles sont observées[38], comme on peut le voir sur la table A.1. Par rapport au *comment*, quelques-unes de ces propriétés peuvent être mesurées ou vérifiées en employant des techniques de vérification statique alors que les autres ont besoin de vérification dynamique et de contrôle au moment de l'exécution. Par rapport au *où*, ces propriétés peuvent être mesurées ou vérifiées dans l'application gérée ou dans le mécanisme d'adaptation.

Table A.1: Classification des propriétés d'adaptation selon *comment* et *où* elles sont observées.

Adaptation Property	Property Verification Mechanism	Where the Property is Observed
Stability	Dynamic	Managed Application
Accuracy	Dynamic	Managed Application
Short Settling Time	Dynamic	Both
Small Overshoot	Dynamic	Managed Application
Robustness	Dynamic	Adaptation Mechanism
Termination	Static	Adaptation Mechanism
Atomicity	Static	Adaptation Mechanism
Consistency	Both	Managed Application
Scalability	Dynamic	Both
Security	Dynamic	Both

Stabilité. Le degré dans lequel le processus d'adaptation va converger vers l'objectif de contrôle (c'est-à-dire l'entrée de référence de contrôle). Une adaptation instable va répéter indéfiniment l'action d'adaptation sans améliorer ou, plus encore, en dégradant l'application gérée à des niveaux inacceptables.

Exactitude. Cette propriété est essentielle pour assurer que les buts d'adaptation sont atteints de manière précise, dans la limite de tolérance donnée. L'exactitude doit mesurer si l'approximation de l'application gérée se rapproche de l'état souhaité (c'est-à-dire, aux valeurs d'entrée de référence).

Temps de reconfiguration court. Le temps requis par le système adaptatif pour atteindre l'état souhaité. Ce temps représente le délai du système pour s'adapter. Il est impossible de définir son acceptabilité en termes absolus pour toutes les domaines d'application du système.

Dépassement de petites ressources. L'utilisation de ressources de calcul pendant le processus d'adaptation pour atteindre l'état souhaité doit être la plus petite possible. Gérer les ressources de dépassement est important pour éviter également l'instabilité du système.

[38]Une propriété est *observable* pour un élément donné d'un système (c'est-à-dire, le *mécanisme d'adaptation* ou l'*application gérée*), indépendamment de qui est le propriétaire de ces élements.

Robustesse par rapport à l'imprévisibilité du contexte. Les services d'application des logiciels gérés doivent rester inaltérées même si l'état de l'application gérée diffère de l'état attendu d'une façon mesurée.

Terminaison (du processus d'adaptation). Dans les approches d'ingénierie du logiciel, le planning MAPE-K produit typiquement un plan d'adaptation sous la forme d'une liste discrète d'actions qui modifie l'application gérée. La terminaison garantit que cette liste soit finie et que son exécution se termine.

Atomicité. Dans le même contexte de terminaison, soit le système s'adapte et le processus d'adaptation se termine avec succès, soit il ne se termine pas et le processus d'adaptation abandonne. Si un processus d'adaptation échoue ou abandonne, le système retourne à l'état cohérent précédent.

Cohérence. Cette propriété cherche à assurer l'intégrité structurelle et comportementale de l'application gérée après avoir effectué un processus d'adaptation. Par exemple, pour une reconfiguration dynamique, cette propriété garantit la connexion de l'interface et les liaisons entre les composants.

Passage à l'échelle. La capacité d'un mécanisme d'adaptation à supporter des demandes de plus en plus importantes de capacités de traitement d'adaptation avec une performance soutenue, en employant des ressources de calcul additionnelles. Par exemple, quand l'adaptation doit évaluer un nombre important de conditions de contexte.

Sécurité. Dans un processus d'adaptation sécurisée, non seulement l'application gérée, mais également les données et les composants partagés avec le mécanisme d'adaptation doivent être protégés en termes de confidentialité, d'intégrité et de disponibilité.

A.3 Modèle Formel

Nous proposons un modèle formel pour la préservation de contrats de QoS lors de la reconfiguration dynamique en combinant deux systèmes formels : la théorie des Machines à États Finis (Finite State Machine ou *FSM* par son acronyme en anglais)[Hopcroft *et al.*, 2006] et la théorie de Systèmes de Transformation de Graphes Typés Attribués (Typed Attributed Graph Trasformation System ou *TAGTS* par son acronyme en anglais, aussi appelés e-graphs) [Ehrig *et al.*, 2009]. Plus précisément, même si cette combinaison s'inspire de la boucle MAPE pour atteindre une autonomie de reconfiguration qui répond à des événements de contexte violant les contrats QoS, nous avons employé ces deux systèmes formels pour modéliser spécifiquement l'élément de planification ou *Planner* de la boucle MAPE.

D'une part, notre choix de *TAGTS* pour modéliser la configuration structurelle du logiciel et sa reconfiguration dynamique nous permettent d'exploiter des patrons de conception au moment de l'exécution. Nous réemployons des patrons de conception déjà existants dans les boucles de reconfiguration pour atteindre les niveaux QoS espérés au moment de l'exécution, de la même façon qu'ils sont employés au moment de la conception. La reconfiguration du logiciel bénéficie alors des propriétés de transformation de graphes. Les patrons de conception représentent un ensemble de composants associés à des opérations de composition , qui permettent de déterminer les niveaux QoS des services auxquels ils participent. D'autre part, nous étendons les *FSM* pour contrôler ces boucles de reconfiguration, pas seulement pour gérer le contrat, mais également pour maintenir l'association de des états avec l'état réel d'exécution du logiciel, soumis aux différentes situations du contexte.

A.3.1 Le modèle basé sur TAGTS pour une Reconfiguration Fiable

Pour obtenir un reconfiguration fiable, nous construisons un modèle sur la théorie TAGTS de façon à définir une architecture logicielle à base de composants (Component-Based Software Structure ou CBS pour son acronyme en anglais) des applications logicielles, des contrats QoS et des règles de reconfiguration. Nous employons ces définitions comme des structures typées afin de garantir la conformité des instances correspondantes et pour spécifier l'opération de reconfiguration dynamique.

Architecture logicielle à base de composants (CBS). Comme l'illustre la Fig. A.2a, nous définissons l'architecture logicielle à base de composants pour une application logicielle comme un nœud de e-graphs qui représentent le composant CBSE usuel, l'interface, le type d'interface et les éléments de liaison. Les composites (aggrégats de composants) s'abstraient comme des composants, au fur et à mesure que nous définissons une reconfiguration structurelle au niveau du système. Les arcs des graphes correspondent aux relations entre ces éléments. Les nœuds d'information représentent les types d'attributs alors que les arcs d'information, les relations typées.

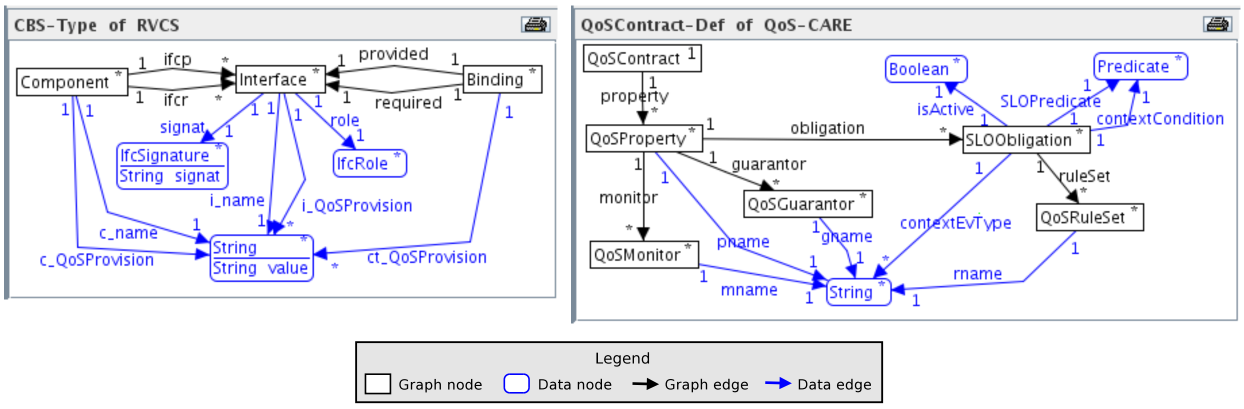


Figure A.2: Représentation graphique d'un e-graph pour (a) les structures CBS (gauche) ; et (b) les contrats QoS (droite).

En gros, une CBS comprend un ensemble d'un ou plusieurs `Components`, avec :

- Un ensemble de Interfaces (`ifcr` et `ifcp`, respectivement), où
- Chaque interface requise peut être unie à une interface fournie au moyen d'un Binding;
- Chaque interface a une signature (`ifcSignature`), un nom (`i_name`) et un rôle (`ifcRole`) ;
- Les composants, interfaces, et unions peuvent avoir des attributs de provision QoS correspondants (`c_QoSProvision`, `i_QoSProvision`, `ct_QoSProvision`, respectivement) pour les noter avec leur provision de capacités QoS particulières, telles que des connexions de réseau sûres ou alors des capacités de processus à échelonner.

Contrats de QoS. Conceptuellement, un contrat de QoS spécifie les niveaux de QoS espérés pour un service d'application ou une fonctionnalité donnée, dans différentes conditions contextuelles. Ces niveaux de QoS deviennent alors des obligations qui peuvent être considérées comme des garanties offertes par un système ou par un fournisseur de services à ses usagers. Toutefois les conditions de contexte respectives sont obtenues au moment de l'exécution. Ainsi, un contrat QoS est un invariant à préserver par un système, par exemple, en le restaurant en cas de transgression. L'évaluation de la validité de l'invariant doit se faire au moment de l'exécution, puisqu'elle dépend du contexte réel d'exécution, tels que le temps de réponse, le débit et le niveau de sécurité des points d'accès au réseau. Comme l'illustre la Fig. A.2b, un contrat QoS se compose d'un ensemble d'éléments `QoSProperty`, où :

- Chaque propriété de QoS possède un ensemble de niveaux de QoS (`SLOObligation`) ;
- Chaque niveau de QoS est associé avec (i) le prédicat correspondant (`SLOPredicate`) pour pouvoir être atteint sous (ii) une condition de contexte correspondante (`contextCondition`) ; (iii) une règle de configuration qui garantit une reconfiguration non-vide (`QoSRuleSet`) à appliquer en réponse à des événements qui appartiennent à (iv) l'ensemble (non-vide) d'événements de contexte (définissant le type `contextEvType`). Les différents niveaux de QoS impliquent des conditions de contexte disjonctives et des ensembles de types d'événements de contexte disjonctifs.
- Un `QoSGuarantor` qui fait référence à un élément du système distinct parmi ceux qui ont la responsabilité d'accomplir les obligations SLO, et
- Un `QoSMonitor`, qui fait référence à un élément du système qui notifie les événements respectifs de contexte.

Règles de reconfiguration. Conceptuellement, une règle de reconfiguration représente une stratégie pour définir un niveau de QoS particulier dans une propriété de QoS. Le type des stratégies qui ont été analysées et proposées dans différents domaines de connaissance des sciences informatiques les définissent comme des *patrons de conception*. Par exemple, Ramachandran et Dougherty *et al.* présentent des catalogues complets de patrons de conception de propriétés de sécurité pour permettre des communications valides de facon transparente ; toutes les communications invalides, que ce soient non-autorisées, non-authentifiées, non-attendues, non-invitées ou non-désirées qui sont bloquées [Ramachandran, 2002, Dougherty *et al.*, 2009]. Pour des systèmes distribués, Krakowiak and Buschmann *et al.* présentent des patrons de conception de *disponibilité* et de *performance* [Krakowiak, 2009, Buschmann *et al.*, 2007]. Comme l'illustre la Fig. A.3, les architectes qui gèrent l'évolution des systèmes peuvent profiter de ces patrons de conception et les coder dans les parties gauche et droite des règles de configuration. Une règle de reconfiguration possède :

- Une partie gauche, qui spécifie les conditions nécessaires qu'une structure gérée doit respecter pour que la règle soit appliquée.
- Une partie droite, qui spécifie la structure du logiciel dans lequel les éléments appariés doivent être reconfigurés.
- Optionnellement, une ou plus de conditions négatives, qui spécifient les conditions qui évitent l'application de la règle.

A.3.2 Sémantique basée sur FSM pour Contrats QoS

Pour atteindre une robustesse face à l'imprévisibilité du contexte et ses implications sur les états des applications gérées, nous avons conçu un FSM qui généralise les états et transitions prévisibles et imprévisibles relatifs à l'accomplissement du contrat QoS. Autrement dit, nous dérivons tous les états atteignables d'une exécution gérée face à tous les événements et conditions de contexte du contrat QoS définis par l'usager. Donc, ce FSM nous permet d'analyser et de raisonner précisément sur ces états et transitions, qu'ils soient spécifiés ou non par l'usager dans un contrat donné.

Sémantique des contrats de QoS. Nous définissons la sémantique de nos contrats de QoS comme une extension de machines à états finis, *QoSC_FSM*. Dans cette extension, les états

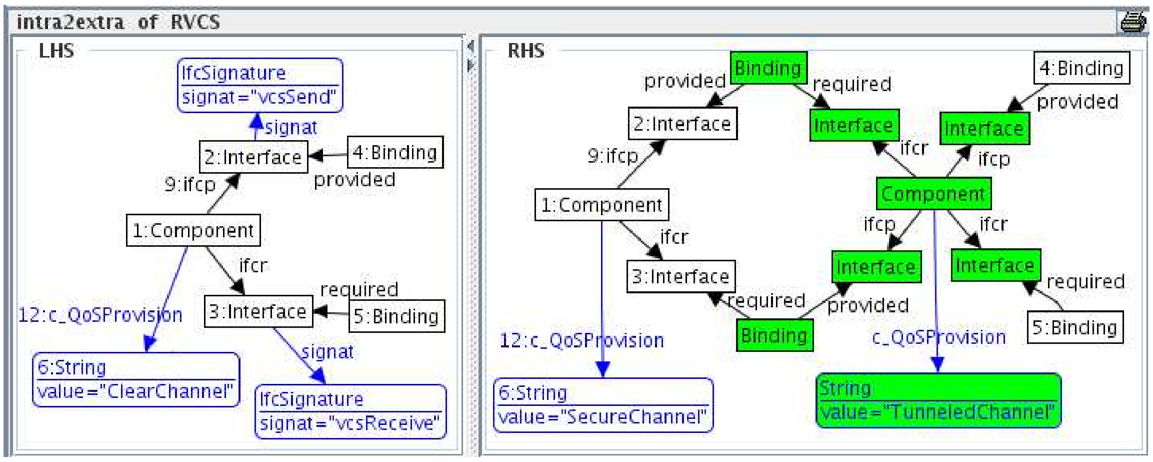


Figure A.3: Représentation graphique e-graph d'une règle de reconfiguration.

représentent les niveaux de QoS espérés et les transitions sont franchies exécutant les opérations de reconfiguration de logiciel. Pour cette sémantique, nous avons généralisé les conditions pour les transitions des états du contrat que ce soit pour l'accomplissement et le non-accomplissement, en atteignant donc, la robustesse relative à l'imprévisibilité du contexte. La figure A.4 montre la forme générale du *QoSC_FSM*, avec trois états et leurs transitions correspondantes, à savoir :

- Contrat Accompli : il résume tous les états d'accomplissement tels qu'ils sont spécifiés dans le contrat.
- Contrat non-Accompli (Exception) : il modèlise les situations dans lesquelles l'usager spécifie, soit :
  - Un ensemble incomplet de règles de reconfiguration (c'est-à-dire, aucune règle ne correspond avec un état gérée ou à une condition de contexte ; soit
  - Une règle ou un ensemble de règles de reconfiguration dont la demande se traduit par une application qui viole les contraintes d'intégrité à base de composants.
  - Un ensemble d'événements contextuels à l'exclusion d'autres qui se produisent réellement da,s le contexte d'exécution de l'application.
- Contrat non-Accompli (Instable) : il modélise la situation dans laquelle les règles spécifiées ne sont pas pertinentes ou pas assez pour atteindre un accomplissement d'un niveau de QoS donné.

Le système de reconfiguration pour préserver un contrat de QoS. Tel que mentionné précédemment, l'objectif final de notre modèle formel est la préservation autonome et fiable de contrats de QoS, soumis à des conditions contextuelles variables à l'exécution. Cette préservation du contrat est défini en termes de cycles de reconfiguration continue qui débutent et finissent pendant l'exécution des applications logicielles gérées, déclenchées par les changements de contexte. Ces cycles de reconfiguration sont contrôlées par notre système de reconfiguration qui préserve le contrat de QoS, définis comme suit :

1. *(Quand reconfigurer).* Une reconfiguration d'application logicielle gérée se déclenche chaque fois que le `QoSMonitor` notifie un événement contextuel qui viole la condition de niveau de QoS actuelle.
2. *(Reconfigurer Quoi, Où et Comment).* À partir de la représentation du contrat, et basé sur l'événement contextuel notifié, la propriété de QoS affectée est identifiée. Puis, la fonction

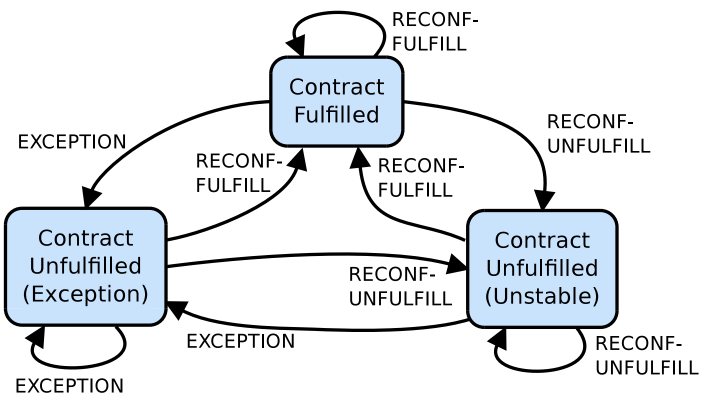


Figure A.4: Machine à états pour le système de reconfiguration qui préserve un contrat de QoS. Dès la conception, le système débute dans un état de contrat-accompli. Les changements de contexte déclenchent les transitions (c'est-à-dire, les reconfigurations).

de transition est appelée, en identifiant l'état cible à atteindre dans la situation du nouveau contexte, avec le prédicat de niveau de QoS correspondant et en garantissant l'ensemble de règles de reconfiguration. Cette fonction réalise un transition d'état, qui reconfigure la structure de l'application gérée. À partir de tout cela, un nouveau plan de reconfiguration est synthétisé.

3. *(Vérifications préalables à l'actualisation).* Une fois la reconfiguration effectuée, la conformité structurelle basée sur les composants est verifiée dans la nouvelle structure de réflexion de l'application gérée. N'importe quelle violation des conditions concernées mène à un état de non-accomplissement du contrat (avec une notification à l'usager).
4. *(Actualisation de la reconfiguration de l'application gérée).* Si la nouvelle structure passe les vérifications préalables à l'actualisation, le plan de reconfiguration synthétisé s'applique à l'application pendant l'exécution. Autrement, il reste non modifié et l'usager est notifié à son tour. Dans tous les cas, l'état du contrat et l'état du logiciel en exécution sont actualisés en conséquence, dans l'état en exécution du *QoSC_FSM*.

A.4 Architecture et implémentation de QOS-CARE

Pour mettre en œuvre notre modèle formel, nous avons conçu une architecture SCA qui crée et maintient une représentation e-graph de l'application gérée et un contrat QoS correspondant à l'exécution. De plus, cette architecture lie la représentation sous forme d'e-graph avec la structure de l'application en exécution réelle en implémentant des fonctions qui maintiennent la cohérence entre les deux modèles structurels. Nous avons conçu ceci à l'aide d'une couche SCA ayant des propriétés de reconfiguration dynamique pour préserver les contrats QoS. Nous l'avons implémenté, déployé et exécuté sur FRASCATI[39], un intergiciel SCA multi-échelle et flexible [Seinturier *et al.*, 2009].

La figure A.5 illustre les composants de l'architecture SCA de QOS-CARE avec ses propriétés exposées et les services fournis et requis. Ces composants sont regroupés selon les éléments de boucle MAPE (étiquetés M, A, P, E et K, respectivement) et ils considèrent les opérations associées de notre modèle formel comme ses services. Les deux propriétés exposées sont le contrat de QoS et l'application logicielle basée sur des composants qui doivent

[39]Nous avons employé FRASCATI v. 1.4 en fixant quelques bugs trouvés lors de l'exécution des primitives de reconfiguration

satisfaire à ce contrat. Pour configurer l'état initial de l'exécution du système entier, la plate-forme d'exécution (par exemple, FRASCATI) charge et exécute QOS-CARE. Le composant `QoSC_FSMManager` demande à FRASCATI de charger l'application spécifiée ; puis il charge le contrat de QoS et il génère la machine à états à partir de ce contrat en utilisant le service `translateTo`*`QoSC_FSM`* du `QoSContractManager`. Ensuite, en employant le service `getCBSAR` de `SCAInstrumentation`, il obtient la représentation SCA de l'application gérée à partir du conteneur de composants FRASCATI au moment de l'exécution et produit la structure de réflexion de l'application à base de composants sous forme de e-graph (c'est-à-dire, `CBSAR`). Avec cette structure, les niveaux de QoS initiaux spécifiés dans le contrat pour chaque propriété QoS, le composant `QoSC_FSMManager` initialise le *QoSC_FSM execution state* (QES).

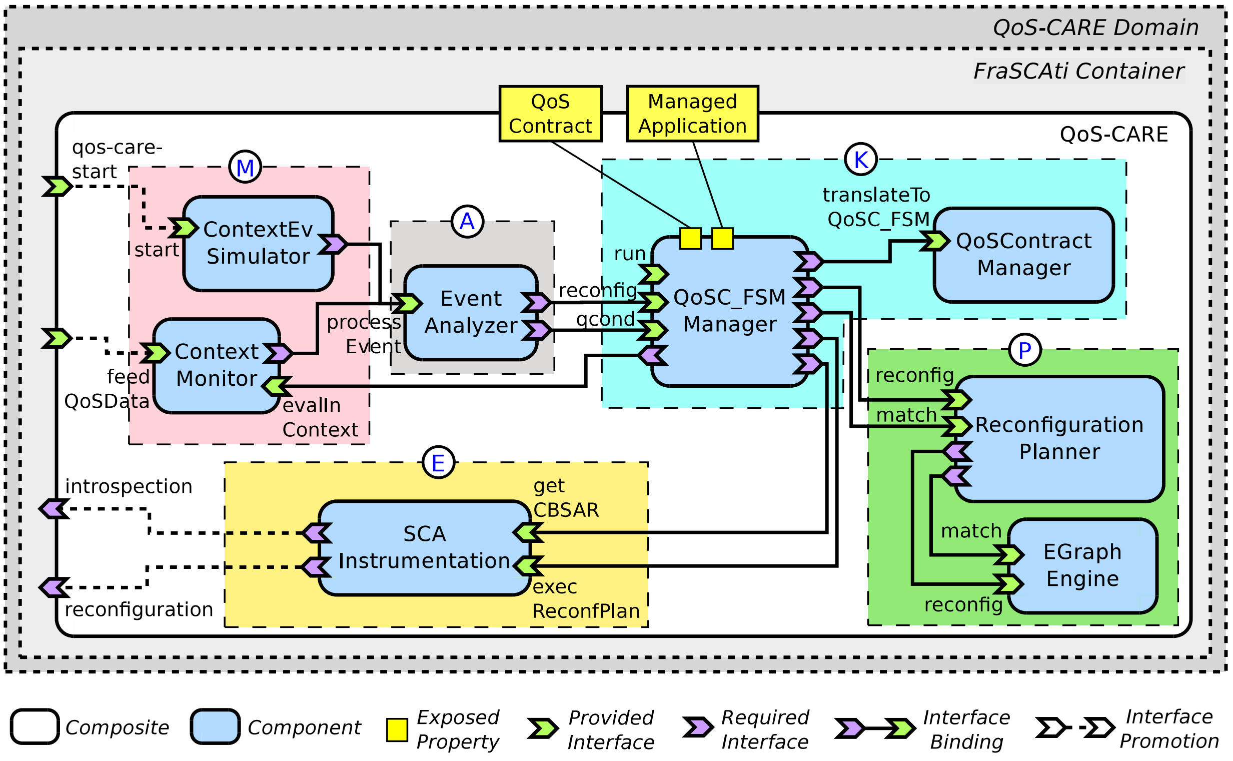


Figure A.5: L'architecture conforme à SCA de QOS-CARE dérivée du modèle MAPE-K.

Une fois que le QES initial est configuré, l'exécution de l'application commence. Parallèlement, le *QoSC_FSMManager* a la responsabilité principale de contrôler les opérations afin de préserver la satisfaction du contrat de QoS spécifié. Le *QoSC_FSMManager*, un composant dérivé de l'élément gérant la connaissance de la boucle MAPE, extrait l'information adéquate à partir du contrat dans le but d'effectuer la reconfiguration souhaitée, en accord avec la situation contextuelle. Cette information correspond à la propriété de QoS affectée, l'état de transition envisagé vers le niveau de QoS, et les règles de reconfiguration à appliquer. Puis, il utilise les règles de reconfiguration pour obtenir la structure reconfigurée, en suivant les étapes décrites précédemment (cf. Section A.3.2, dans le système de reconfiguration pour préserver le contrat QoS).

A.5 Validation

Jusqu'à présent, nous avons présenté les deux contributions principales de cette thèse. La première étant l'identification et la définition des *propriétés d'adaptation*. La deuxième, une solution globale au problème de la reconfiguration autonome et fiable de logiciels basés sur des composants en vue de préserver les contrats de QoS soumis à des conditions d'exécution variables.

Pour la validation de la première contribution, nous avons présenté l'identification et la définition des *propriétés d'adaptation* à la communauté SAS, en ouvrant une discussion sur l'analyse de ces définitions, leur adoption et standardisation. Quant à la deuxième contribution, au fur et à mesure que nous concevions notre solution comme une architecture et une implémentation SCA, nous l'avons intégré pour sa validation comme une couche supplémentaire dans le middleware FRASCATI. Puis, nous l'avons appliqué dans deux cas d'études, nous avons validé théoriquement ses propriétés et nous avons effectué une évaluation expérimentale de ses performances. Le premier cas d'étude est un système de visioconférence mobile avec un contrat de QoS qui garantit la *disponibilité* et la *confidentialité* des services de visiconférence. Le deuxième est une simple application Web de mash-up qui compose et orchestre dynamiquement le service de localisation du réseau social Twitter[40] avec un service web météo de différents fournisseurs d'information météorologique, tels que Google[41] et WebServiceX[42]. Sur la base de la satisfaction d'un contrat de QoS en rapport avec le *niveau de prévision* du service météorologique, la disponibilité du service est garantie. Les résultats positifs de l'évaluation positive ont confirmé la faisabilité, l'applicabilité et la (ré)utilisabilité de notre système de reconfiguration.

Validation et vérification des propriétés d'adaptation : Temps de conception vs. Temps d'exécution. Dans cette thèse nous définissons la fiabilité du processus de reconfiguration compte tenu de cinq propriétés d'adaptation : temps court de configuration, terminaison, atomicité, conformité structurelle SCA et robustesse relatifs à l'imprévisibilité du contexte. Dans la table A.2, nous résumons *quand* (temps de conception vs. temps d'exécution) et *où* (mécanisme de reconfiguration vs. Application gérée) que nous avons verifiés et validés.

Table A.2: *Quand* et *où* se valident et se vérifient les propriétes de QOS-CARE.

	Design-time V&V	Run-time V&V
Reconfiguration Mechanism	• Atomicity • Robustness	• Settling-time • Termination
Managed Application	(Not Applicable)	• SCA Structural Conformance

Grace à notre modèle formel, les propriétés d'atomicité et de robustesse sont validées dans le mécanisme de reconfiguration, étant donné que leurs conditions sont suffisantes pour être vérifiées au moment de la conception. En revanche, les conditions nécessaires pour garantir les propriétés de terminaison et conformité structurelle à SCA, présentes dans notre modèle formel, doivent être vérifiées au moment de l'exécution dans leurs instances correspondantes. C'est-à-dire, qu'elles doivent être vérifiées sur les règles de reconfiguration donnés par l'utilisateur dans le cas de terminaison (c'est-à-dire dans le mécanisme de reconfiguration) ; et sur la représentation e-graph réelle du système à l'exécution dans le cas de la conformité structurelle à SCA (c'est-à-dire dans le logiciel gérée). Nous mesurons le temps de reconfiguration experimentalement en faisant une moyenne des valeurs respectives des temps d'exécution, en employ-

[40]`https://dev.twitter.com/docs/api`
[41]`http://code.google.com/p/java-weather-api`
[42]`http://www.webservicex.net/ws/WSDetails.aspx?CATID=12&WSID=56`

ant des références de performance sur le mécanisme de reconfiguration pour notre domaine d'application, comme une garantie raisonable pour le temps de reconfiguration prévu.

A.6 Conclusion

Nous classons les contributions de cette thèse en deux parties, selon nos deux objectifs principaux. Dans la première partie, nous avons caractérisé les propriétés inhérentes aux systèmes logiciels auto-adaptatifs (SAS). En plus d'être utiles au moment de l'évaluation des mécanismes d'adaptation en modes comparables et standardisés, ces propriétés peuvent être utilisées pour améliorer et combiner ces mécanismes. Les propriétés d'adaptation que nous proposons pour la standardisation sont davantage importantes dans le sens où elles peuvent aider à mettre en avant les bénefices de l'auto-adaptation, pas seulement pour sa plus ample adoption dans l'industrie, mais également pour la discipline du génie logiciel en soi. Par exemple, les métriques correspondantes de ces propriétés peuvent fournir un aperçu de la fiabilité de façon à incorporer ces mécanismes dans l'automatisation partielle des phases de maintien et d'évolution des cycles de vie des logiciels.

Dans la deuxième partie, nous avons suivi une stratégie globale pour définir QoS-CARE, qui est un mécanisme de reconfiguration robuste et fiable pour préserver les contrats de QoS dans les applications logicielles à base de composants. Cette stratégie comprend quatre aspects, centrés sur l'élément de planification du modèle de la boucle de rétroaction MAPE-K : un modèle formel, sa réalisation comme une architecture logicielles SCA, l'implémentation de cette architecture et sa validation à la fois théorique et expérimentale. Suivant cette stratégie, nous avons mis en œuvre des modèles (formels) au moment de l'exécution pour reconfigurer fiablement et robustement des applications logiciels de façon à préserver leurs contrats de QoS.

Par conséquent, avec QoS-CARE, nous avons (i) montré la faisabilité d'exploiter des patrons de conception à l'exécution dans des boucles de rétroaction pour accomplir les niveaux de QoS associés à des conditions contextuelles spécifiques, et (ii) fourni une couche supplémentaire middleware SCA pour préserver de façon fiable les contrats de QoS dans des applications logicielles à base de composants.