Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Περιβάλλον-Πλαίσιο για την Αυτόματη Αναδιαμόρφωση Συστημάτων Λογισμικού

Διπλωματική Εργασία

του

Κωνσταντίνου Αθανάσιου Αθανασίου

Επιβλέπων: Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Σεπτέμβριος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Περιβάλλον-Πλαίσιο για την Αυτόματη Αναδιαμόρφωση Συστημάτων Λογισμικού

Διπλωματική Εργασία

του

Κωνσταντίνου Αθανάσιου Αθανασίου

Επιβλέπων: Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Σεπτεμβρίου, 2014.

....	....	....
Κώστας Κοντογιάννης	Ιωάννης Βασιλείου	Γεώργιος Στάμου
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2014

....
Κωνσταντίνος Αθανάσιος Αθανασίου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © – All rights reserved Κωνσταντίνος Αθανάσιος Αθανασίου, 2014.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Τα πολύπλοκα σύγχρονα συστήματα λογισμικού όπως αυτά του ηλεκτρονικού εμπορίου, των υπηρεσιών νέφους και των ηλεκτρονικών τραπεζικών υπηρεσιών, λειτουργούν υπο συνεχώς μεταβαλλόμενες προτεραιότητες χρηστών, περιβάλλοντα, και διαθέσιμους πόρους. Σε αυτό το πλαίσιο, οι πάροχοι υπηρεσιών λογισμικού επιθυμούν να παρέχουν ελαστικές και αποκριτικές υπηρεσίες και παράλληλα να διατηρούν τα λειτουργικά τους έξοδα ελάχιστα. Αυτοί οι αντικρουόμενοι στόχοι οδηγούν σε υπέρογκα συστήματα λογισμικού που είναι δύσκολα να διαχειριστούν ακόμα και από ειδικούς πληροφορικής. Ώς εκ τούτου, αυτο-διαχειριζόμενα συστήματα τα οποία μπορούν να αποκρίνονται στο περιβάλλον τους έχουν γίνει αναγκαία.

Σε αυτή την εργασία, παρουσιάζουμε ένα περιβάλλον πλάισιο για την αυτόματη αναδιαμόρφωση συστημάτων λογισμικού. Η προσέγγισή μας υλοποιεί έναν αυτόνομο βρόχο ελέγχου και χρησιμοποιεί μοντέλα στόχων υψηλού επιπέδου για να ορίσει στρατηγικές επίλυσης προβλημάτων που προκύπτουν κατά τη διάρκεια λειτουργίας του συστήματος. Ένα πλάισιο συλλογιστικής αναγάγει το πρόβλημα σχεδιασμού λύσης στο πρόβλημα της προτασιακής ικανοποιησιμότητας και συντασει ένα πρόγραμμα αναδιαμόρφωσης το οποίο εφαρμόζεται στο σύστημα. Τέλος, παρουσιάζουμε ένα πρωτότυπο του περιβάλλοντος πλαισίου και το εφαρμόζουμε σε υποδομές νέφους, οι οποίες μοντελοποιούνται με χρήση του προσομοιωτή CloudSim.

Λέξεις Κλειδιά

autonomic computing, προσαρμοστικά συστήματα, δέντρα στόχων, planning, προτασιακή ικανοποιησιμότητα, τεχνολογία λογισμικού

Abstract

Contemporary complex IT systems such as e-commerce, cloud services and e-banking are required to operate under constantly changing user priorities, environments, and available resources. In this context, vendors wish to provide resilient and responsive services while in the same time keeping their operational costs to a minimum. These conflicting factors lead to systems that are too big for even the most skilled administrators to manage. Therefore, systems that are able to respond to their environment in the form of self-adaptation and manage themselves according to administrator goals have become a necessity.

In this thesis, we present a framework for self-adaptive system reconfiguration. Our approach implements the autonomic control loop and uses high-level Goal Models to define solutions to possible issues that may arise during the execution time of the system. A reasoning framework reduces the planning problem to the propositional satisfiability problem and compiles a reconfiguration plan which is effected on the system. Finally, we provide a concrete implementation of the proposed framework and apply it on top of Cloud IT infrastructure, which is modeled using the CloudSim simulation framework.

Keywords

autonomic computing, self-adaptive systems, goal models, planning, propositional satisfiability, software engineering

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου εργασίας κ. Κώστα Κοντογιάννη για την εμπιστοσύνη που μου έδειξε και για την καθοδήγηση του καθ΄ όλη την διάρκεια εκπόνησης της διπλωματικής μου εργασίας. Επίσης θα ήθελα να ευχαριστήσω τους διδακτορικούς φοιτητές Γιώργο Χατζηκωνσταντίνου και Μιχάλη Αθανασόπουλο για τις συμβουλές τους καθώς και για την παραχώρηση κώδικα που αποτελεί τμήμα δικής τους έρευνας, και όλα τα μέλη του εργαστηρίου τεχνολογίας λογισμικού για το ευχάριστο κλίμα συνεργασίας.

Ευχαριστώ τους συμφοιτητές και φίλους μου για τις γνώσεις που αποκομίσαμε και τα ωραία χρόνια που περάσαμε.

Τέλος, ευχαριστώ την οικογένειά μου και τη σύντροφό μου για την συμπαράσταση τους κατά τη διάρκεια των σπουδών μου και τη στήριξή τους σε κάθε μου επιλογή.

Κωνσταντίνος

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Information technology, being a competitive and on demand business, always seeks to improve the resilience and responsiveness of its services, while in the same time keeping its operational costs at a minimum level. These driving factors along with contemporary complex software and hardware solutions (e-commerce, cloud services, e-banking etc.) which are required to operate under constantly changing user priorities, environments, and available resources, fuel the problem of managing such systems.

The majority of today's computing hardware and software infrastructure tackle this issue with low-level mechanisms (i.e. exceptions, fault-tolerant protocols etc.) which have a limited view of the system, are application specific, and costly to implement and later modify. Additionally, this approach depends on well-trained human operators, increases the maintenance costs, is prone to error and is unable to scale up to the contemporary multi-layered and complex IT infrastructure.

It follows naturally that the aforementioned needs, push both the research community and the industry towards devising systems that are able to respond to their environment in the form of *self-adaptation*. This notion introduces the idea that systems are able to decide in an autonomous manner on how to accommodate changes in their environment. By autonomous we mean that they can either act without any human interference or with some guidance in the form of high-level objectives (i.e. policies).

The concept of self-adaptation brings design decisions of a system towards runtime in order to control dynamic behaviors, and enforces it to reason about its state and environment. For example, keeping the utilization of a physical host of a cloud data-center at high levels requires a) the collection of information that showcase the current state of the host (e.g metrics that prove the host's low utilization), b) the analysis of this information to diagnose possible issues (e.g. correlation of multiple performance metrics), c) decision making on how to approach this issue(e.g. targeting future workloads on this host) and d) actions that will take in effect the decisions previously made.

By observing the example previously presented, one can realize that a promising starting point towards self-adaptation are techniques originating from control engineering and natural systems[1]. Thus, feedback loops are able to provide a generic mechanism for self-adaptation. A feedback loop will typically involve four states; collect, analyze, decide, and act, as shown in figure 1.1. The feedback cycle starts with the collection of relevant data from environmental sensors and other sources that reflect the current state of the system. Next, the system analyzes the collected data. Next, a decision must be made about how to adapt the system in order to reach a desirable state. Finally, to implement the decision, the system must act via available actuators or effectors.

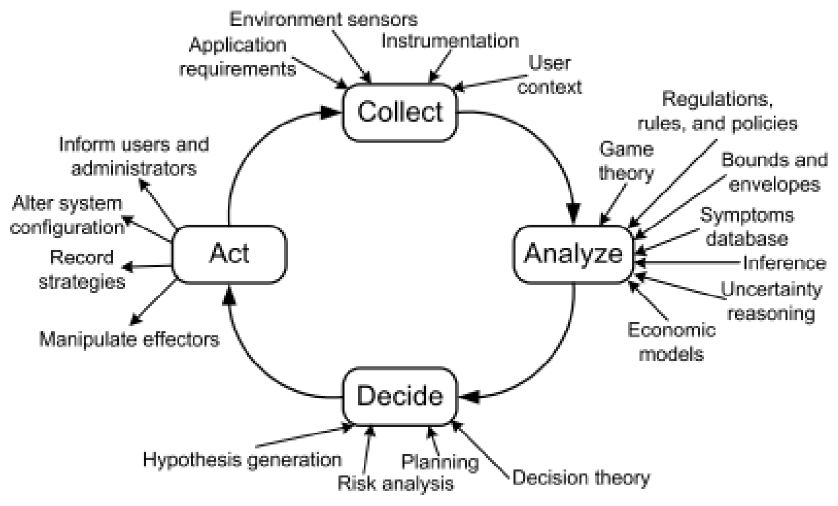


Figure 1.1: Autonomic Control Loop

Kephart and Chess[2] proposed the first architecture that implemented such a feedback loop in IBM's architectural blueprint for autonomic computing[3], which is further discussed in Section 2.1. Garlan et al[4] have also proposed the addition of an external controller to the underlying system, essentially implementing a feedback loop. At this point it is important to discern, that the common factor amongst such initiatives that aim to achieve self-adaptation, is the need for software that supports the proper engineering and realization of self-adaptation. Building such systems in a cost-effective and predictable manner is a major software engineering challenge that only recently has seen the first attempts to establish suitable software engineering approaches for the provision of self-adaptation. The encouraging results of the research community formed around this subject have established self-adaptive systems as an active and interdisciplinary research field.

1.2 Problem Description

The vision of creating IT systems that are able to manage themselves according to administrators' goals[2], brings with it many open challenges for the Software Engineering community. [5] and [6] aim to summarize and identify such research challenges that arise when developing, deploying and managing self adaptive systems, in the following four axes:

i. *Design space:* What decisions a developer should address when designing a self-adaptive software system

ii. *Processes*: How to define generic processes for the development, deployment, operation, maintenance, and evolution of self-adaptive systems.

iii. *Decentralization of control loops*: How to approach control loops for various degrees of centralization and decentralization of the controlled elements.

iv. *Practical run-time verification and validation*: how to obtain information regarding trust in self-adaptation.

In this thesis we visit the first two categories of problems. In the context of *design space*, bridging the gap between the design and the implementation of self-adaptive systems constitutes a major challenge. Frameworks and reference architectures can be of significant assistance in this problem by providing reusable models and robust infrastructure to developers of self-adaptive systems. They can also contribute in the process of creating self-adaptive on top of non-self-adaptive systems, by helping developers to better understand the interaction of the control loop and other self-adaptation mechanisms with the non-self-adaptive system and how actions of the one system affects the other.

In the context of *processes*, the key research challenge is the design of a generic framework for self-adaptive software systems that provides reasoning support for the system's adaptation based on relative costs and benefits of its actions. Such a framework should include (i) the appropriate infrastructure and support for the definition of such actions and the dependencies amongst them; (ii) support for the concrete implementation of these actions in the framework's context; (iii) a reasoning and analysis mechanism for the decisions that the system must take in order to execute them, based on their costs and benefits.

The main goal of this thesis is the design and implementation of a framework for self-adaptive systems able to reason for its decisions according to the benefits and costs of the actions needed to stabilize its performance. The framework must a) implement the feedback loop showed in figure 1.1 and provide robust interfaces for the concrete implementations of all the phases of the loop, b) make use of expressive, domain independent models that will allow it to decide about self-adaption in an intelligent manner, and c) be able to reason about these models in an efficient way.

High-performance heterogeneous distributed systems such as data-centers, clusters and grid computing systems constitute a killer application domain for self-adaptive systems due to the multi-dimensional goals and objectives that they wish to meet. For example, a vendor providing cloud services, needs to be able to automatically detect and remediate reduced performance of its systems in order to provide to its customers the agreed quality of service (QoS). This could be translated to addition of extra computing resources so that the vendor can satisfy its customer's needs. However such a decision works in contrast to the vendors general policy for better power management and reduction of operational costs. Hence, such complex and multi-parametric problems present excellent challenges for self-adaptive computing.

For our proposed framework to be tested, we have built our self-adaptive system on top of CloudSim[7], a simulation framework for cloud infrastructure which is further discussed in Section 2.4.2. CloudSim has been extended in order to produce and channel performance metrics to our system, which monitors CloudSim's execution. When an abnormal behavior is observed by our system, the latter creates an alert and its reasoning mechanism is triggered in order to remediate the monitored issue. For our system to decide how to react, we model solutions to possible issue as collections of *Task Resolution Models*, which are formally introduced in Section 5.1. The reasoning mechanism responds with a compilation of actions that need to be effected on CloudSim, and finally these actions are executed after taking in account the current state of CloudSim.

1.3 Contributions

In this thesis we present the design and implementation of a framework for self-adaptive systems with emphasis on plan compilation and execution. Our contributions are summarized as follows:

- Design of a domain independent reference architecture for self-adaptive systems. The architecture follows the blackboard architectural model[8] and implements the autonomic control loop[9]. The architecture assumes the usage of Task Resolution Models for its analyze part.

- Definition of a Task Resolution Model that extends the goal models utilized in Requirements Engineering (and further explained in section 2.2) by associating each entity with a cost/benefit value and by adding contextual constraints. An instance of our model represents solutions to possible runtime issues and shortcomings that the self-adaptive system might face during its execution, as AND/OR trees.
- Transformation of the aforementioned Task Resolution Model to propositional formulas and reduction of the planning problem to Boolean Satisfiability.
- Design of an algorithm that guarantees the retrieval of all the parts of our Task Resolution Model that are relevant to the monitored issue.
- Extension of the CloudSim framework in order to (i) support dynamic workloads for memory, network and storage metrics; (ii) provide Key Performance Indicators of its resources.
- Prototype implementation of the self-adaptive framework based on the proposed architecture, and application of the latter on top of the CloudSim simulation framework.

1.4 Outline

The rest of the thesis is structured as follows:

Chapter 2 introduces background knowledge that is related to our work. The basic concepts of autonomic computing and self-adaptive systems are introduced along with a reference architecture defined by IBM. Following these, background information and definition of Goal Models and some of their variations is presented. Next, we introduce decision procedures and propositional logic, as the formal system that is employed for our systems reasoning process. A discussion is held on the boolean satisfiability decision problem, its algorithms and some of its extensions. Next, we provide some basic background on cloud computing and finally we present different approaches for modeling cloud infrastructure.

Chapter 3 presents the designed self-adaptive framework. Section 3.1 showcases the high level architecture of the system, its core modules and how it related to IBM's MAPE-K reference architecture. Section 3.2 introduces the component diagram of our proposed reference architecture for a self-adaptive framework. Next, each component of the reference architecture is visited and its subcomponents, and interfaces are described in detail. When needed, the interaction of the components with their subcomponents or other components is explained through sequence diagrams. Finally, section 3.3 provides a sequence diagram showcasing a session of the basic adaptation loop that the framework implements.

Chapter 4 presents the domain models of the self-adaptive framework. The core entities of the framework and their relations are visited. Additionally, the touchpoints of the self-adaptive system are explained, and finally a domain model of the extensions implemented to the CloudSim framework, which plays the role of the controlled system, is discussed.

Chapter 5 presents in detail the reasoning framework of the self-adaptive system. First we introduce the models utilised and formally define them. Then we provide an algorithm capable of reducing the size of the model that the framework has to reason on, in order to devise an adaptation plan. Finally, we show the reduction of Plan derivation to boolean satisfiability and the construction of the cnf formula fed to the sat solver.

Chapter 6 presents experimental evaluation of the proposed framework on top of the CloudSim simulation framework. Section 6.1 describes the execution setup of our implemented self-adaptive system and the simulation characteristics of the CloudSim framework. Furthermore, in the context of the self-adaptive system, its concrete implementations of the extensible part of the framework are discussed. Section 6.2 showcases experiments run for two different use cases, comparing the behavior of the the CloudSim framework, when the latter is run in a non - adaptive and in an adaptive setup.

Chapter 7 concludes our work and provides directions for future work.

Chapter 2

Related Work

2.1 Autonomic Computing - Adaptive Systems

2.1.1 Basic Concepts

Autonomic computing[2] is a term introduced by IBM in 2001 proposing a computing environment able to manage itself and dynamically adapt to changes occurring in its context in order to meet certain objectives and policies. This process is automatic and minimizes the involvement of an IT professional, essentially hiding the complexity of the system from operators and users. Both the term and the motivation for autonomic computing are derived from human biology, as the human body executes most of its vital tasks i.e. breathing, in a self-managing manner without any conscious effort from the person itself.

To achieve this autonomic behavior, system components (such as operating systems, hypervisors, storage units etc) need to be able to act in a self-managing manner by observing their environment and taking appropriate actions according to the situation. These components - hereby autonomic elements - are enhanced with control loop functionality which can be organized in the following four categories:

- *Self-Configuring:* Enables the component to adapt to its changing environment. By sensing changes on the deployment of the system's components, removal of components or critical changes to a component's state, a self-configuring component utilizes policies provided by an IT professional to guarantee strength and productivity of the IT infrastructure.
- *Self-Healing:* Enables the component to monitor, analyze and react to upheavals. After detecting a system malfunction the component enables corrective actions

according to predefined policies, without disrupting the IT environment. Such actions usually involve altering the state of the component or even effecting changes in other components of the environment.

- *Self-Optimizing:* Enables the automatic tuning of resources. A self-optimizing component will monitor its performance and tune itself in order to meet end-user business requirements. To do so, it proceeds in actions such as reallocation of resources to secure high utilization and high standards of service both for the system and the end user.

- *Self-Protecting:* Enables the detection, identification and protection of threats directed to the component from the environment. A *self-protecting* component can detect hostile activities and take corrective actions, or even act in a proactive manner to make itself less vulnerable. Examples of such hostile behaviors are unauthorized accesses, denial-of-service attacks and these components ensure the conservation of security and privacy policies.

IBM has also proposed a reference model for control loops [3] called the `MAPE-K` loop which is depicted in figure 2.1 and is used to communicate the architectural aspects of autonomic systems. It represents how a single entity is managed in an autonomic environment and forms an autonomic element. The managed resource is the controlled system component and it can either be a single or a collection of resources. The *sensors* and *effectors* provide mechanisms to collect information about the state of the managed resource and to change its state respectively, and are analyzed in greater detail in the next section. The autonomic manager is the component that implements the control loop which is dissected in four parts, namely *Monitor*, *Analyze*, *Plan*, *Execute* while constantly consulting and updating its *Knowledge* of the system.

Autonomic elements do not act independently one from another and may cooperate to achieve common goals [3], e.g. servers in a cluster optimizing the allocation of resources to applications to minimize the overall response time or execution time of the applications. Consequently autonomic managers need not be aware only of the condition of their controlled elements, but also of the condition of other autonomic managers in their environment. Both multi-agent approaches[2] and hierarchical compositions of autonomic elements[10] have been proposed.

Although the control loops of these self- * properties share some fundamental parts, they all aim to secure different attributes that are needed for a system to automate its tasks. Systems management software is responsible for the system wide orchestration of tasks defined by these control loop functionalities. To make this process possible,

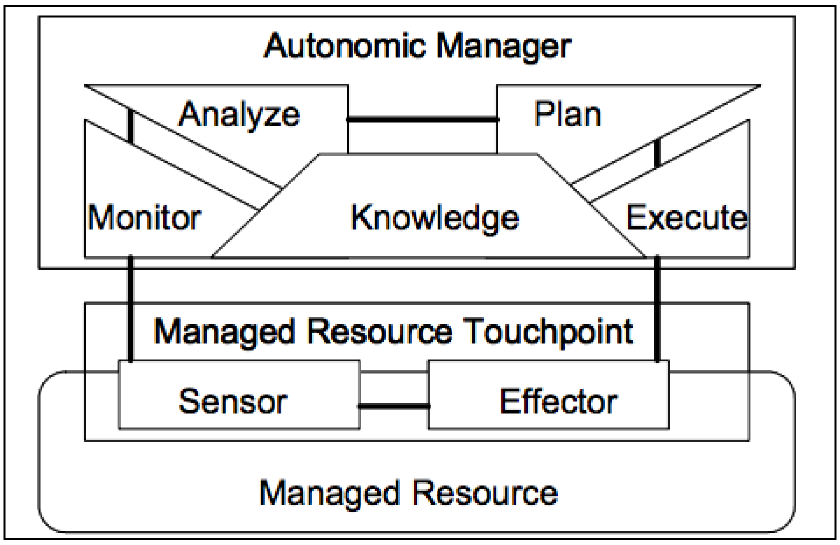


Figure 2.1: MAPE-K Loop

IT processes delegation is required in the sense that an autonomic manager, i.e. a self-optimizing manager, is assigned to a single component, i.e. a server, and is responsible for optimizing this server's performance only.

The addition of control loop functionality to system components cannot be instant and requires gradual, multi-stage evolution of enterprise infrastructure. To better understand and plan this process, IBM has proposed the following five levels of autonomicity[11]:

Basic (Level 1)
This level is the one in which most contemporary systems belong to. All the components of the system are manually installed and managed by IT professionals. The latter are also responsible for monitoring, aggregating and analyzing the systems' environment and data.

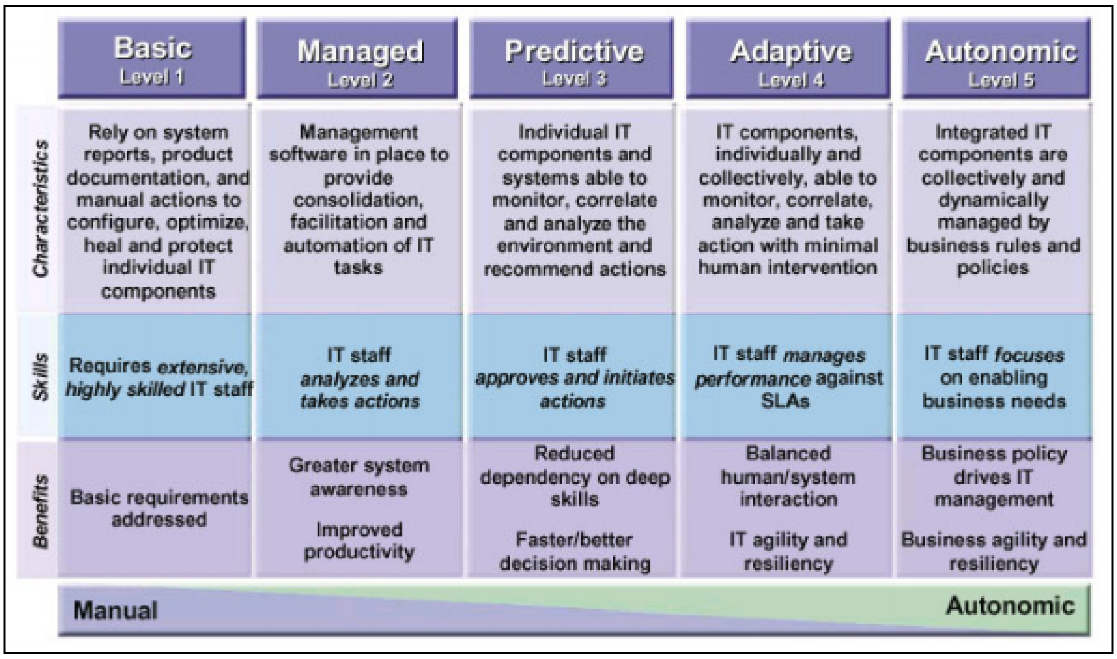
Managed (Level 2)
This level is reached by systems that aggregate data and actions from multiple components into a main central unit by usage of management tools. IT professionals are responsible for the analysis of data and execution of task, benefiting from the vast amount of information that the system provides.

Predictive (Level 3)
This level includes systems that are able to monitor and correlate data in order to recognize patterns and support the IT professionals with recommended tasks and actions. Management integration across multiple components is achieved, allowing the system to be managed by professionals with less technical skills.

Adaptive (Level 4)
This level goes one step beyond the predictive level by not only correlating, developing and suggesting plans, but also executing plans according to policies. Consequently the IT infrastructure at this level is more responsive to changing business conditions, allowing the organization to balance between human and system interactions.

Autonomic (Level 5)
The final level describes well integrated components, able to dynamically manage themselves according to rules and policies while communicating with each other. At this level the staff focuses on the definition of business requirements, which are the main managing tools of the IT infrastructure.

An overview of the levels of autonomicity is given at figure 2.2.



	Basic Level 1	Managed Level 2	Predictive Level 3	Adaptive Level 4	Autonomic Level 5
Characteristics	Rely on system reports, product documentation, and manual actions to configure, optimize, heal and protect individual IT components	Management software in place to provide consolidation, facilitation and automation of IT tasks	Individual IT components and systems able to monitor, correlate and analyze the environment and recommend actions	IT components, individually and collectively, able to monitor, correlate, analyze and take action with minimal human intervention	Integrated IT components are collectively and dynamically managed by business rules and policies
Skills	Requires *extensive, highly skilled* IT staff	IT staff *analyzes and takes actions*	IT staff *approves and initiates actions*	IT staff *manages performance* against SLAs	IT staff *focuses* on enabling business needs
Benefits	Basic requirements addressed	Greater system awareness Improved productivity	Reduced dependency on deep skills Faster/better decision making	Balanced human/system interaction IT agility and resiliency	Business policy drives IT management Business agility and resiliency

Figure 2.2: Autonomicity Levels

2.1.2 Architecture Concepts

In their Autonomic Computing blueprint, IBM proposed an autonomic computing reference architecture depicted in figure 2.3. The bottom part of the architecture contains the managed resources of the autonomic system that constitute the hardware and software of the IT infrastructure. It is possible for some of these components to already guarantee some basic self-managing capabilities. The next level includes the touchpoints to the IT infrastructure which serve as an interface to control and manage the system's resources. The two following levels are primarily responsible for the automation of the system and contain the autonomic managers. At level three, an autonomic manager is responsible for the control of a single resource, i.e. a database server, and it can implement one of the four control loop types described above (self-configuring, self-healing, self-optimizing, self-protecting). Level four includes autonomic managers whose responsibility is the orchestration of the autonomic managers of the previous level and they are able to implement control loops taking into account a broad view of the system. Finally the top layer includes a manual manager as an interface for IT professionals which provides system management capabilities. All five layers are able to obtain and share information via knowledge sources, as shown in figure 2.3. The building blocks of the reference architecture are described next.

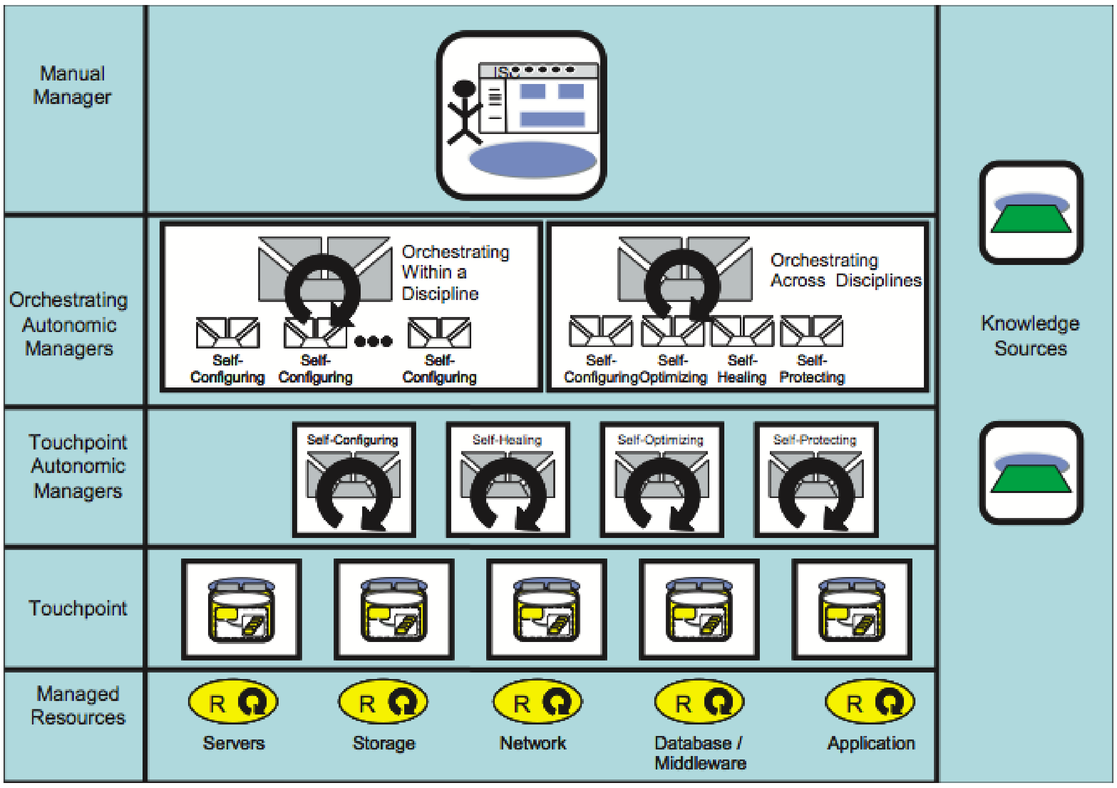


FIGURE 2.3: Autonomic System Layers

Autonomic Managers

Autonomic managers are systems components responsible for automating some management functionality of the system by implementing the intelligent `MAPE-K` loop and publishing its results through management interfaces. The internal structure of autonomic managers is organized by the autonomic computing reference architecture in the following four sections:

- **Monitor:** The monitor function collects information produced by the managed resources via touchpoints, aggregates them and models them as symptoms that can be analyzed. This information involves metrics, configuration details, setting etc. The data can be static, meaning that they remain the same or barely change through time, and dynamic, changing during the systems execution. Data is aggregated, filtered and correlated for the monitor function to determine a symptom that is related to a particular combination of events. Afterwards, this symptom is passed to the analyze function. Due to the colossal amount of data that is processed from the touchpoint sensor, modeling and interpreting them in an efficient manner is of paramount importance for an autonomic manager.

- **Analyze:** The analyze function observes and analyzes symptoms provided by the monitor function in order to decide whether corrective actions should be applied. This is achieved by ensuring that the autonomic manager is conformant with the current policies and goals. Violation of these, triggers the reasoning part of the analyze function, which by employing data analysis and reasoning techniques on the observed symptoms and consulting the knowledge data, issues a request for change to the plan function. This request contains all the information that exhibit the modifications, that according to the analyze function, need to be applied to the system in order the remediate the monitored symptoms. The complex analysis mechanisms allow the autonomic manager to better understand and model the IT environment leading to easier prediction of future behaviors and thus better symptom diagnosis.

- **Plan:** The plan function is responsible for creating a procedure or compiling multiple ones in a complex workflow, enactment of which will cause the desired alteration to the system. After the workflow is created it is sent to the execute function to implement the desired changes to the managed resource.

- **Execute:** The execution function enacts the plans created by the plan function to the system through a series of action. The actions are carried out by utilizing the touchpoint effector of managed resources. Finally the execution part may be responsible for updating the knowledge of the autonomic manager.

Knowledge Sources

Knowledge sources are repositories in which data such as symptoms, policies, change requests and plans are stored and shared among autonomic managers. Knowledge has particular types and semantics and can be used by all four functions of an autonomic manager. Finally, autonomic managers of autonomic managers can also obtain knowledge from the latter in order to perform additional management tasks.

Manual Manager

A manual manager involves the implementation of the user interface that allows an IT professional to execute management function in a manual manner. The manual can potentially work with other autonomic managers that are on the same level of orchestration or control autonomic managers and other IT professionals working at lower levels.

Touchpoints

Touchpoints are system components that expose interfaces called *manageability interfaces* able to query the state of managed resources as well as to execute management operations on managed resources. Therefore a touchpoint is the implementation of a manageability interface for a single resource (e.g. a virtual host) or a set of related managed resources (e.g. a database server, the databases hosted on the server and the tables of the databases).

The *manageability interface* of a managed resource is organized into its **sensor** and **effector** interfaces. The **sensor** interface provides two different types of interactions: first get-type operations that provide information about the current state of the managed resource, and second events that follow a send-notify fashion which are issued when the managed resource reaches states that are worth reporting. The **effector** interface respectively, provides the two following types of interactions: set-type operations that change the state of the managed resource, and events that enable the managed resource to issue requests to its autonomic manager.

2.2 Goal Models

Goal Models have been extensively utilised in Requirements Engineering to model early requirements[12] and more recently for behavioral customization of software systems[13]. Goal Models can express high level goals that stakeholders wish to achieve and capture alternative ways in which these goals can be met.

2.2.1 Goal Trees

Loosely speaking, a Goal Model is a set of goal trees whose nodes are connected by contribution links. Goal trees consist of goal nodes which are AND/OR decomposed to subgoals, meaning that if a goal G is AND-decomposed (OR-decomposed) to subgoals $G_1, G_2, \ldots, G_n$ then if all (at least one) of its subgoals are satisfied, G is also satisfied. A contribution link is a lateral connection between two goal nodes, expressing how fulfillment or denial of the source goal affects the target goal. The intuitive meaning of the contribution link $G_1 \xmapsto{++_S} G_2$ ($G_1 \xmapsto{++_D} G_2$) is that given G_1 is satisfied (denied), G_2 is satisfied (denied) in return. The meaning of the $--_S, --_D$ links is dual w.r.t. the outcome of the target goal. Figure 2.4 depicts an example of a Goal Model.

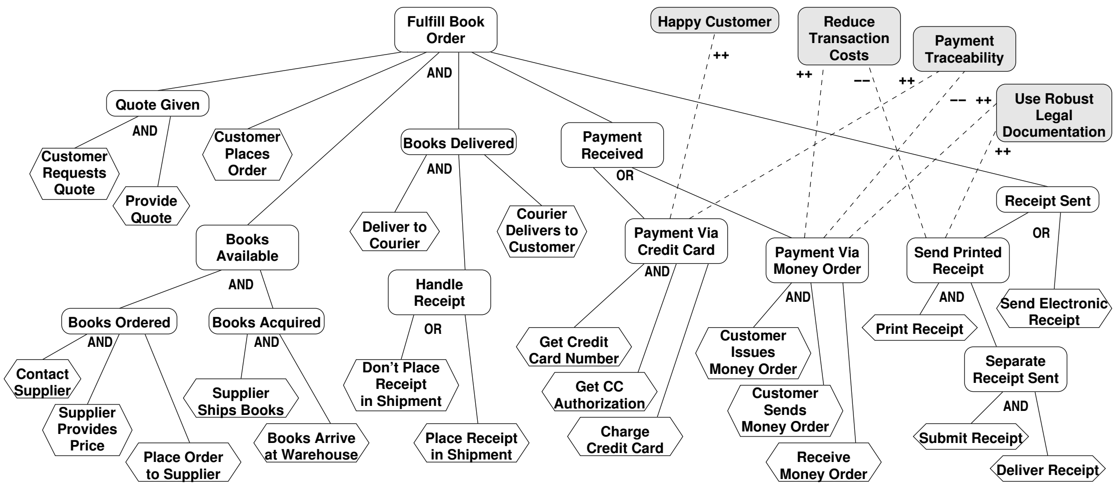


FIGURE 2.4: Sample Goal Model

At this point it is important to distinguish the difference between decomposition and contribution; decomposition expresses intentionality regarding the achievement of a goal node while contribution works as a side effect. For example, for goal *Books Available* to be satisfied one must first assure satisfiability of goals *Books Ordered* and *Books Acquired* to which the first is AND-decomposed to. On the other hand, satisfiability of the source goal node *Send Printed Receipt*, leads to lateral denial of node *Reduce Transaction Costs* as a result of their $--_S$ relation.

A Goal Model can be formalized as a tuple $\langle \mathcal{G}, \mathcal{R} \rangle$ where $\mathcal{G}$ is the set of goals and $\mathcal{R}$ is the set of relations among goals. A relation r over goal nodes G_i is denoted as $(G_1, \ldots, G_n) \overset{r}{\mapsto} G$, where $G_1, \ldots, G_n$ are the *source goals* and G is the *target goal* of r. Boolean relations are the n-ary *and*, *or* relations while contribution relations are the binary $++_S, ++_D, --_S, --_D$ relations. A *root goal* is any goal with an incoming boolean relation and no out coming ones and a *leaf goal* is any goal with no incoming boolean relations. Finally, every Goal Model has to conform to the following two restrictions:

- each goal has at most one incoming boolean relation
- every loop contains at least one non-boolean relation arc

The propagation of satisfiability and deniability through a Goal Model is formalised by the axioms presented in table 2.1.

Relation	**Axioms**
$(G_1, G_2) \xmapsto{and} G :$	$(G_1 \wedge G_2) \rightarrow G$
	$\neg G_1 \rightarrow \neg G$
	$\neg G_2 \rightarrow \neg G$
$(G_1, G_2) \xmapsto{or} G :$	$G_1 \rightarrow G$
	$G_2 \rightarrow G$
	$(\neg G_1 \wedge \neg G_2) \rightarrow G$
$G_1 \xmapsto{++S}:$	$G_1 \rightarrow G_2$
$G_1 \xmapsto{--S}:$	$G_1 \rightarrow \neg G_2$
$G_1 \xmapsto{++D}:$	$\neg G_1 \rightarrow \neg G_2$
$G_1 \xmapsto{--D}:$	$\neg G_1 \rightarrow G_2$

Table 2.1: Satisfiability Axioms

2.2.2 Contextual Goal Models

Traditional Goal Models do not take in account contextual variability and how the latter can affect the modeled system. To tackle this problem in contemporary high-varying ubiquitous computing systems, various Requirements Engineering approaches that introduce contextual variability in Goal Models have been suggested [14],[15]. Contextual Goal Models aim to identify under which set of circumstances parts of the Goal Model are present and to provide a version of the Goal Model that is conformant to this set.

To define Contextual Goal Models, let E be the set of elements in a Goal Model, and T be the set of element types (e.g. goal node, decomposition, etc.). A function M maps each element of E into an element of T relating each element of the Goal Model to a type. The set $T^C \subseteq T$ includes the types of elements that are affected by contexts. Finally the set $E^C = \{n \mid n \in E \wedge M(n) \in T^C\}$ contains the elements which are context dependent. Finally, the set C contains the contextual tags that are assigned to members of E^C representing the conditions under which these elements are part of the Goal Model. Each tag $c \in C$ is assigned a boolean expression $p : p \rightarrow c$ which defines when this tag is active. A special tag def that is active by default is also defined, and is associated with these elements that are visible in the Goal Model regardless of context.

An element is possibly associated with multiple sets of tags e.g. $m \in M^C$ is assigned the set of tags $\{\{a, b\}, \{c, d\}\}$ to indicate that m is visible either in the case that a and b are active or in the case that c and d are active. Tags are implicitly propagated from *target goals* to *source goals* due to the hierarchical structure of Goal Models reducing substantially the number of tags used in a Contextual Goal Model. Examples of Contextual Goal Models are given in figure 2.5.

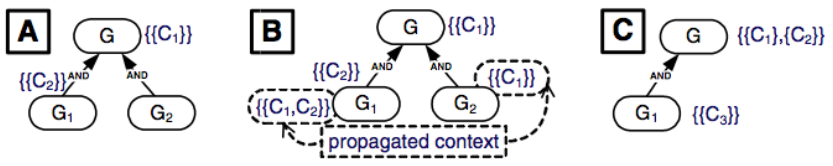


FIGURE 2.5: Sample Contextual Goal Model

Model elements are contained in the default context which is always active ($\{def\}$). To express that for the goal G to be achieved the context C_1 must be active the tag $\{\{C_1\}\}$ is attached to G in figure 2.5 A. If a goal can be achieved when either one of the assigned tags are active, multiple sets of tags have to be used, like the set $\{\{C_1\},\{C_2\}\}$ assigned to G in Fig 3.5B. An example of the tag propagation is give in figure 2.5 B where the tag C_1 applied to G is also applied to G_1 and G_2. Finally, tags are combined when used in the same subtree, as is the case of figure 2.5 C. Goal G can be achieved when either C_1 or C_2 are active while G_1 when C_3 is active. Due to G_1 being part of the subtree of G, the combined tags are $\{\{C_1, C_3\},\{C_2, C_3\}\}$.

2.3 Decision Procedures

A **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters and a **decision procedure** is an algorithm that, given a decision problem, terminates with a correct yes-or-no answer[16].

In the context of this thesis, the planning part of our autonomic system is modeled as a decision procedure. A decision procedure in our system would answer to the question *'Given the current state of the system, a set of policies that apply to the latter and a set of possible repair actions, is there a possible combination of actions that could be executed in order to remediate the monitored issue?'*. For our purposes, we will look into the **Boolean Satisfiability** decision problem (**SAT**) and some of its extensions such **Satisfiability Modulo Theories** (**SMT**), and **Weighted MAX-SAT**. Before doing so, we introduce basic knowledge on the propositional logic formal system.

2.3.1 Propositional Logic

Logic was initially studied by the ancient Greeks in order to analyze the laws of reasoning and was considered to be part of philosophy. The 1800's was the period when modern mathematical logic was born and created many questions to mathematicians, who had to create what later would become computer science.

Propositional logic is a simple yet powerful fragment of logic[17] where expressions can only have two values, namely F and T. The language of propositional logic consists of:

1. The constant expressions $true$ and $false$ which always evaluate to T and F
2. A set of primitive symbols (usually p, q and r) referred to as atoms, propositional letters or variables that range over the values of T and F
3. A set of operator symbols interpreted as logical operators or logical connectives

We proceed by visiting some logical connectives.

- *Negation* $\neg$ is a unary operator and we say that $\neg p$ is the negation of atom p and is thought of as its denial. When p is true, $\neg p$ is false; and when p is false, $\neg p$ is true. $\neg\neg p$ always has the same truth-value like p.
- *Conjuction* $\wedge$ is a binary connective between propositions p and q, is denoted by $p \wedge q$ and expresses that each of the propositions is true. For any two propositions its meaning is given by the truth table 2.2.
- *Disjunction* $\vee$ is also a binary connective between two proposition p and q, is denoted by $p \vee q$ and expresses that either p or q are true. It must be noted that while in spoken language *or* often means p or q but not both, in propositional logic *or* always means at least one. This difference becomes obvious in table 2.2.
- *Implication* $\Rightarrow$, also called material or logical implication, is denoted by $p \Rightarrow q$ where p is the antecedent and q the consequent, and expresses that q is true when p is true. Implication is usually confused with causation, however implication only relates two propositions by their truth-values and does not express a cause and effect relation. To better understand this difference, it is useful to remember that $p \ \Rightarrow \ q$ is equivalent to $\neg \ p \ \vee \ q$.
- *Equivalence* $\Leftrightarrow$ is denoted by $p \ \Leftrightarrow \ q$ and expresses that p and q have the same value.

p	**q**	$\neg$**p**	**p** $\wedge$ **q**	**p** $\vee$ **q**	**p** $\Rightarrow$ **q**	**p** $\Leftrightarrow$ **q**
F	*F*	*T*	*F*	*F*	*T*	*T*
F	*T*	*T*	*F*	*T*	*T*	*F*
T	*F*	*F*	*F*	*T*	*F*	*F*
T	*T*	*F*	*T*	*T*	*T*	*T*

TABLE 2.2: Truth table of logical connectives

Truth tables can be used not only to define the connectives, but also to test for valid sentences. Given a propositional formula, by constructing a truth table for the formula and looking at the column of values obtained we can characterize it as:

- *satisfiable* if there is at least one T
- *unsatisfiable* if it is not satisfiable, i.e., all entries are F
- *falsifiable* if there is at least one F
- *valid* if it is not falsifiable, i.e., all entries are T

Finally, a formula is in **Conjunctive Normal Form (CNF)** or **clausal normal form** if it is a conjunction of clauses. A *clause* is a disjunction of literals, for example a 3-clause is a disjunction of exactly 3 literals (e.g. $p \vee q \vee \neg r$). A *literal* is a proposition symbol or its negation (e.g. p or $\neg\, p$). For example $(a \vee b) \wedge (\neg b \vee c \vee \neg d) \wedge (d \vee \neg e)$ is a CNF formula with three clauses and three literals. As a normal form, it is utilised in automated theorem proving and it serves as the input format of modern SAT solvers.

2.3.2 Boolean Satisfiability and Extensions

A natural question that arises is whether we can check the satisfiability of a propositional formula. A simple algorithm would be to construct the formula's truth table and afterwards easily decide satisfiability and validity. However the computation time of this approach is exponential to the number of proposition symbols and therefore impractical.

Satisfiability or **SAT** is the decision problem of determining if there exists an interpretation that satisfies a given propositional formula. Differently put, it checks if there is an assignment over the variables of the formula that evaluates to T. If no such assignment exist, the formula is identically F and is called *unsatisfiable*, and *satisfiable* if such an assignment exists. Cook showed that **SAT** is NP-complete [18] - the first known example of an NP-complete problem - meaning that there is no known algorithm that efficiently solves all instances of **SAT**.

The variety of problems appearing in applications such as model checking, automated planning and scheduling, and diagnosis in artificial intelligence that can be transformed into instances of the **SAT** decision problem, make it a hot research topic. A class of algorithms called *SAT solvers* that can efficiently solve a large enough subset of SAT instances, is extremely useful in such applications.

The *DPLL algorithm*[19] is a complete, backtracking based search algorithm for deciding the satisfiability of propositional formulas and serves as the basis of the most efficient

SAT solvers. The algorithm chooses a literal, assigns a truth value to it, simplifies the formula and then recursively checks if the simplified formula is satisfiable. If it is not, the same recursive check is executed, assuming the opposite truth value for the literal. The efficiency of the algorithm greatly depends on the choice of the literal which is considered in the backtracking step, known as the *branching literal*. Consequently the DPLL algorithm includes a family of algorithms, one for each strategy used for the choice of the branching literal. These strategies are known as branching heuristics, and are a field of active research. Implementations of the DPLL algorithm include Chaff[20], zChaff[21] and MiniSat [22], while in the context of this thesis we utilize SAT4J[23], which is based on MiniSat.

The DPLL algorithm has also been applied to **Satisfiability Modulo Theories** (**SMT**) which generalizes **Boolean Satisfiability** (**SAT**) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories[24]. Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. Differently put, one can view SMT as an instance of SAT where some of the binary variables are replaced by predicates over a suitable set of non-binary variables.

Initial implementations of SMT solvers would translate the SMT problem to a boolean SAT instance and then pass it to a Boolean SAT solver. The main advantage of this *eager approach* is that by transforming to an equivalent Boolean SAT formula one can leverage the performance of already existing SAT Solvers. However this approach ignores the more expressive, high-level semantics of the underlying theory, and forces the SAT Solver to try hard in order to discover obvious facts. The *lazy approach* intends to integrate the DPLL sat solving algorithm with theory-specific solvers. For this integration to work, the theory solver must be able to infer new facts from already established ones, and must be able to identify conflicts on the theory when the latter arise. Essentially, the theory solver must be incremental and backtrack-able.

The Partial Weighted MaxSat problem is a generalization of the satisfiability problem and is useful in situations where one seeks the maximum number of constraints that can be satisfied with a minimal penalty. The main idea is that sometimes not all restrictions of a problem can be satisfied, so they are divided in two categories: the restrictions or clauses that must be satisfied (hard), and the ones that may or may not be satisfied (soft). Soft clauses can have different weights, representing the penalty for falsifying them and showcase that not all restrictions are equally important. The addition of weights to clauses makes the instance weighted, and the distinction of hard and soft clauses makes the instance partial. Given a weighted partial MaxSat instance we want to find the assignment that satisfies the hard clauses so that the sum of the weights of the falsified clauses is minimal.

The weighted partial MaxSat problem is a NP-hard problem, for which state-of-the-art solvers have not yet experienced the same success as SAT solvers. In our work, we solve the weighted partial MaxSat problem by utilizing the SAT4J solver which is based on successive calls to a SAT solver[25]. This type of MaxSat solvers seem to have better performance for industrial planning problems and have become very competitive for the industrial categories in the MaxSat evaluation.

2.4 Cloud Computing

2.4.1 Basic Concepts

Cloud computing is a model that enables the on-demand access to a shared pool of configurable computing resources such as servers, networks, storage, applications and services which can be provisioned and released with minimal effort[26]. The National Institute of Standards and Technology (NIST), along with the the former definition has discerned the following *essential characteristics* of cloud computing: *on-demand self-service, broad network access, resource pooling, rapid elasticity* and *measured service.*

Furthermore, NIST has divided the services provided by cloud computing into the three following *service models.*

Infrastructure as a Service(IAAS): At this layer the consumer is able to provision computing, storage, network and other fundamental resources which can include operating systems, hypervisors etc. To deploy their applications, cloud users install operating-system images and their application software on the cloud infrastructure.

Platform as a Service(PAAS): At this layer the consumer is able to deploy to the cloud infrastructure applications that are created using programming, languages, frameworks, tools, libraries and services which are supported by the cloud provider. The consumer has no direct access to the underlying infrastructure such as servers, operating systems etc, but has control over the deployed applications and can configure some settings of their environment.

Software as a Service(SAAS): At this layer the consumer has only access over the provider's applications which are run on the cloud infrastructure. The applications can be accessed from client interfaces such as a web browser or even from programming interfaces. There consumer is unable to manage the infrastructure and can only in some cases configure some application specific settings.

As described in the NIST cloud definition, cloud infrastructure can be distinguished by the way it is operated and is divided into four **deployment models**. A *public cloud* is available to the public over a public network and is usually owned by an organization providing cloud services. A *private cloud* is infrastructure that is exclusively used by a single organization and is operated by either the organization itself or a third party. A *community cloud* is used by a community of users or organizations with shared concerns and vision and it can be owned, managed and operated by one or more of the affiliated organizations, a third party or a combination of them. In the case of the *hybrid cloud* the infrastructure is a composition of two or more distinct cloud infrastructures (possibly following different deployment models) that remain unique entities, but are bound by technology and mechanisms that allow data and application portability (e.g. cloud bursting for load balancing between cloud).

2.4.2 Modeling

As promising as the cloud computing model is, it reveals multiple challenges[27] for enterprises, one of them being decision making about management of the infrastructure. This is due to the effects of cloud computing on the work of IT departments and other parts of the organization as well as due to the costs, risks and benefits of running systems in the cloud. In order to help managers to make migration decisions, new tools, techniques and models need to be developed.

Developing a system modeling framework that can be used to model relevant attributes of large-scale IT systems, such as the system's infrastructure and executing applications, would provide multiple benefits, like easier calculation of risks and assistance at migration decisions. These modeling frameworks are relieved from the inherent rigidity of the infrastructure making actions such as benchmarking the application performance under variable conditions or reproduction of results that can be relied upon, less painful.

For this work we have looked into two different approaches to modeling cloud infrastructure. First we visit CloudML[28], a domain-specific modeling language (DSML) to model the provisioning and deployment of multi-cloud systems, and CloudSim[7], a cloud simulator.

CloudML

CloudML makes use of model-driven engineering (MDE) methods and techniques, in order to tackle the heterogeneity and incompatibility of the solutions provided by cloud providers. These MDE methods aim to facilitate the specification of provisioning, deployment, monitoring and adaptation concerns of multi-cloud systems at design-time and their enactment at run-time.

Specifically CloudML provides:

- a Domain-specific modeling language (DSML) for modeling the provisioning and deployment of multi-cloud systems at design-time
- a models@run-time[29] environment for enacting the provisioning, deployment, and adaptation of these systems, as well as monitor their status at run-time. This run-time environment can be accessed by a reasoning system through a model-based interface.

CloudML is agnostic to any development paradigm and technology, meaning that the developers can design and implement the applications based on their preferred paradigms and technologies. It considers two possible roles in the deployment work-flow: a *cloud-app developer* who knows the internals of the systems and can model their requirements, constraints and dependencies, and a *cloud-app vendor* who does not know about the internal of the systems but can specify some requirements, constraints and dependencies according to some policy (i.e. budget constraints). The cloud-app developer initially specifies one or more templates of the provisioning and deployment model and afterwards the cloud-app vendor adjusts and combines these templates into the actual provisioning and deployment model.

CloudML considers provisioning and deployment models at two levels of abstraction, namely Cloud Provider-Independent Model (CPIM), and Cloud Provider-Specific Model (CPSM), as shown in figure 2.6. The CPIM represents a generic provisioning and deployment model that is independent of the cloud provider. The CPIM is transformed semi-automatically into a CPSM, which represents a specific provisioning and deployment model that is dependent on the cloud provider. At run-time, the CPSM is causally connected to the running system; i.e., a change in the CPSM is reflected on-demand in the running system, whereas a change in the running system is automatically reflected in the CPSM.

The models@run-time environment of CloudML which facilitates reasoning about dynamic adaptation of running systems by providing an abstract representation of the

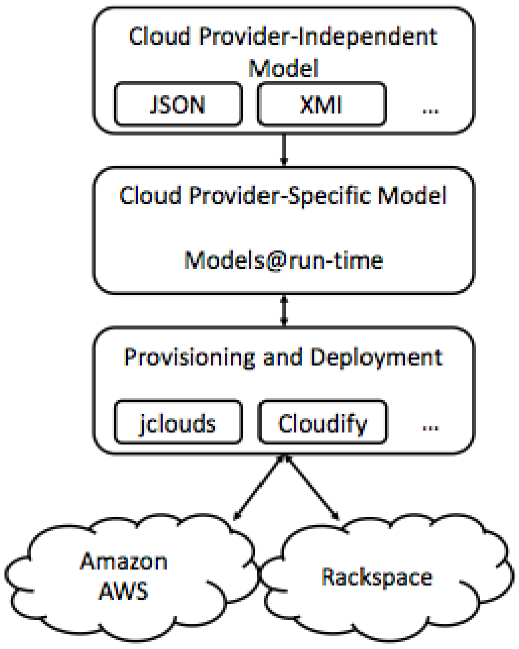

Cloud Provider-Independent Model
JSON
XMI
...
Cloud Provider-Specific Model
Models@run-time
Provisioning and Deployment
jclouds
Cloudify
...
Amazon AWS
Rackspace

FIGURE 2.6: CloudML Architecture

system causally connected to the running system, make it a cloud modeling tool capable to incorporate the self-* properties of an autonomic system.

CloudSim

CloudSim is a generalized and extensible simulation framework that allows modeling, simulation, and experimentation of emerging Cloud computing infrastructures and application services[7]. CloudSim enables it's users to easily test the performance of newly developed application services in a fully control and easy to set up environment. The two main advantages of using CloudSim is (i) *time effectiveness*, as minimal time and effort is required to implement the cloud-based application provisioning and (ii) *flexibility*, as developers can model and test their applications in heterogeneous Cloud environments (Amazon, EC2, etc) without any deployment effort and cost.

CloudSim offers the following features:

- support for modeling and simulation of large scale Cloud computing environments, including data centers, on a single physical computing node
- a self-contained platform for modeling Clouds, service brokers, provisioning, and allocation policies

- a virtualization engine that aids in creation and management of multiple, independent, and co-hosted virtualized services on a data center node
- flexibility to switch between space-shared and time-shared allocation of processing cores to virtualized services

At figure 2.7 we present the multi-layered architecture of the CloudSim framework. The *CloudSim* simulation layer is responsible for the modeling and simulation of virtualized Cloud data-center environments and includes management interfaces for virtual machines, applications (hereafter Cloudlets) as well as cloud resources such as memory, storage and bandwidth. Fundamental actions such as resource provisioning and Cloudlet execution are handled at this layer. This layer also exposes functionalities that a cloud developer can extend to perform complex workload profiling and application performance study, by extending the Cloudlet entity.

At the *User code* simulation layer the characteristics of the cloud resources that are simulated are provided by the user. For hosts, the user provides the number of machines, their specification etc., for Cloudlets the number of tasks and their requirements, for VMs, the number of cores, users, and broker scheduling policies.

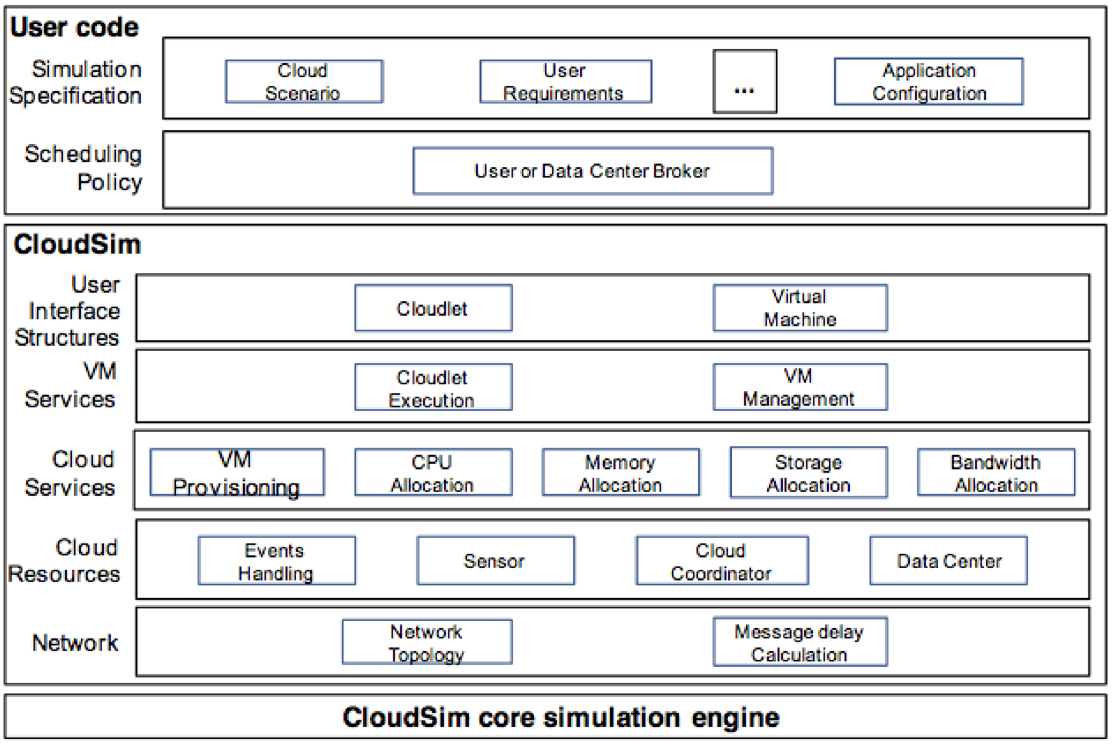


FIGURE 2.7: CloudSim Architecture

For our implementation of an autonomic cloud system, we decided to utilize the CloudSim framework for (i) its extensible Java based entities that allowed us to modify some of their functionality, such as workload requests, Cloudlet configurations and provisioning techniques and are further discussed in Chapter 5; (ii) the abstraction it provides by relieving development from using resources from cloud vendors.

Chapter 3

System Architecture

In this chapter we present the reference architecture of the proposed framework for self-adaptive systems. In parallel we discuss a concrete implementation of this architecture that we developed in order to add self-adaptation to the CloudSim framework.

3.1 Architecture Overview

The key features that frameworks should provide to a developer are *modularity*, *extensibility*, *reusability* and *inversion of control*[30]. Since we are devising a framework for self-adaptive systems, a feedback loop that controls the self-adaptation process is an additional cornerstone feature that our framework must provide[1]. Additionally, in our approach we consider that our self-adaptive system interacts with the controlled system in a **Master/Slave** pattern[6]. The latter introduces a hierarchical relationship between one master (the self-adaptive system constructed with our proposed framework) that is the sole responsible for the analysis and planning part, and one or more slaves(the CloudSim framework) who along with the master are responsible for the monitoring and execution phase of the feedback loop. This hierarchical relationship imposes that our framework is designed in a way that enforces the seamless attachment of an existing system to the framework.

Our architecture is built on top of IBM's architecture for autonomic elements and implements the MAPE-K loop shown in figure 2.1. Furthermore, the architecture of our framework follows the Blackboard architectural pattern[8]. The driving idea of this architectural pattern is that a collection of experts collaborate on shared information in order to provide a solution to a problem which in general has no closed approach.

We proceed by grouping the core modules of our framework according to their role in the MAPE-K loop. The high level architecture of our framework when applied on top of the CloudSim framework is shown in figure 3.1.

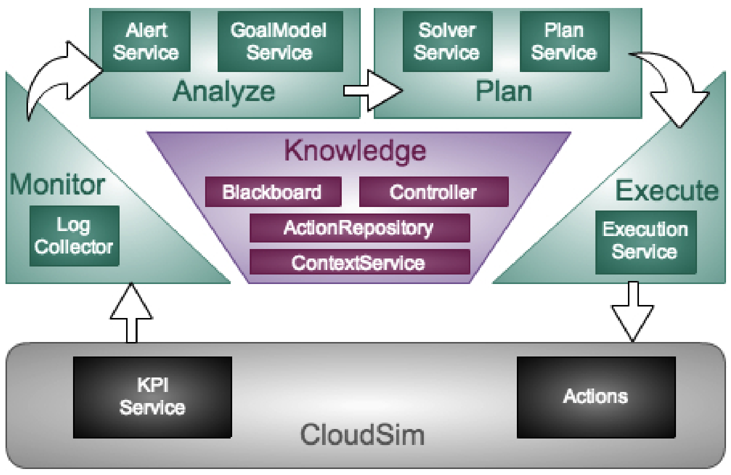


FIGURE 3.1: High Level Architecture

3.1.1 Knowledge

Following IBM's architectural guideline for autonomic elements, knowledge components implement repositories that provide access to knowledge regarding the self-adaptive and the controlled system to the other components of the architecture, through well defined interfaces. The following modules are associated with the concept of knowledge in our framework:

Blackboard: This module serves as the main storage unit of our framework. Along with the Controller module it orchestrates the adaptation loop by receiving events and data generated by the experts and by providing access to this information to other modules/experts who require them.

Controller: This module directs the execution of the adaptation loop. It monitors the blackboard module and notifies accordingly the experts about the former's state and contained data. It is then up to the experts to decide whether they can extend the knowledge contained in the blackboard by executing its functionality and publishing its results.

***ContextService*:** This module provides the framework with information relevant to the controlled system that are able to influence the decision making process of the adaptation loop. Such information can include policies, parameters or runtime constraints of the controlled system. In the case of CloudSim, contextual constraints can be allocation policies (i.e. allocate a specific type of resource) and costs/benefits (i.e. the benefit of migrating a virtual machine to a host that consumes less power or the cost of providing worse quality of services to a customer than the agreed service level agreements).

3.1.2 Monitor

The following module is associated with the monitor function of the MAPE-K loop:

***LogCollector*:** This module is responsible for the accumulation of logging data generated by the controlled system, and thus, strongly interacts with the controlled system's probes. The LogCollector module processes the log data and structures them in entities of the type *LogEvents*. LogEvents are then sent to the blackboard module where they are stored and are available for utilization by other experts during the adaptation loop. LogCollector is considered an expert not only due to the fact that it provides information that serve as intermediate results to the problem of self-adaptation but also because it can exploit the current state of the blackboard (i.e. when the Blackboard contains identical LogEvents with little to no semantic difference) to provide different solutions to the subproblem it is responsible for (i.e filtered data).

3.1.3 Analyze

The diagnosis and modeling functionality of the analyze function is distributed between the following modules:

***AlertService*:** This module includes the components required for the diagnosis and identification of non-desired states that the controlled system has reached and for the upraisal of *Alerts* that model these problematic states. The AlertService expert is notified by the Controller module when LogEvents are stored in the Blackboard and uses them as input for its alert generation process. This process can be simple such as setting thresholds for specific metrics of the controlled system and raising Alerts whenever they are surpassed or complex, like identifying patters of LogEvents that evident some abnormal behavior. Once an alert is generated by the AlertService, it is the stored in the blackboard, as intermediate data that the GoalModelService will utilize in the future.

***GoalModelService*:** This module is responsible for all operations related to Goal Models in our self-adaptation framework. Such operations involve storing and retrieving the Goal Model, modifying it according to contextual constraints that apply during execution time and creating a *Hypothesis* that our system needs to prove. The GoalModelService expert acts upon being notified about alerts that have been issued by the

AlertService module and then delivers the Hypotheses that it generates to the Blackboard, as partial knowledge towards finding a plan that will change the system's state and remediate the initially monitored issue. Finally, changes that occur in the context of the self-adaptive system and which are introduced by the ContextService need to be reflected as changes to the Goal Models by the GoalModelService.

3.1.4 Plan

The plan function of the MAPE-K loop that structures actions into plans in order to meet goals and objectives, is implemented by the following modules:

***SolverService*:** This module encapsulates the formal reasoning of the self-adaptive framework and contains all the different classes of algorithms that can provide solutions over its formal system. Propositional and first order logic are examples of formal systems that a self-adaptive system could employ, while SAT and SMT solvers can provide the framework with solutions of their respective formulas. The SolverService expert operates on top of Hypotheses that are pushed in the Blackboard and after proving that the Hypotheses can be satisfied, it publishes its results back to the Blackboard. Results come as models that satisfy the formula provided by the Hypothesis and represent the set of actions that need to be executed on the controlled system.

***PlanService*:** This module incorporates the functionality related with the compilation of actions included in the solved Models into executable plans. The PlanService expert operates with the solved Models as input and is responsible for evaluating the feasibility of the proposed models, ranking their effectiveness, combining them into more complex execution plans and finally, based on its results, deciding if they should be applied to the controlled system. The final verdict of the PlanService is published to the Blackboard.

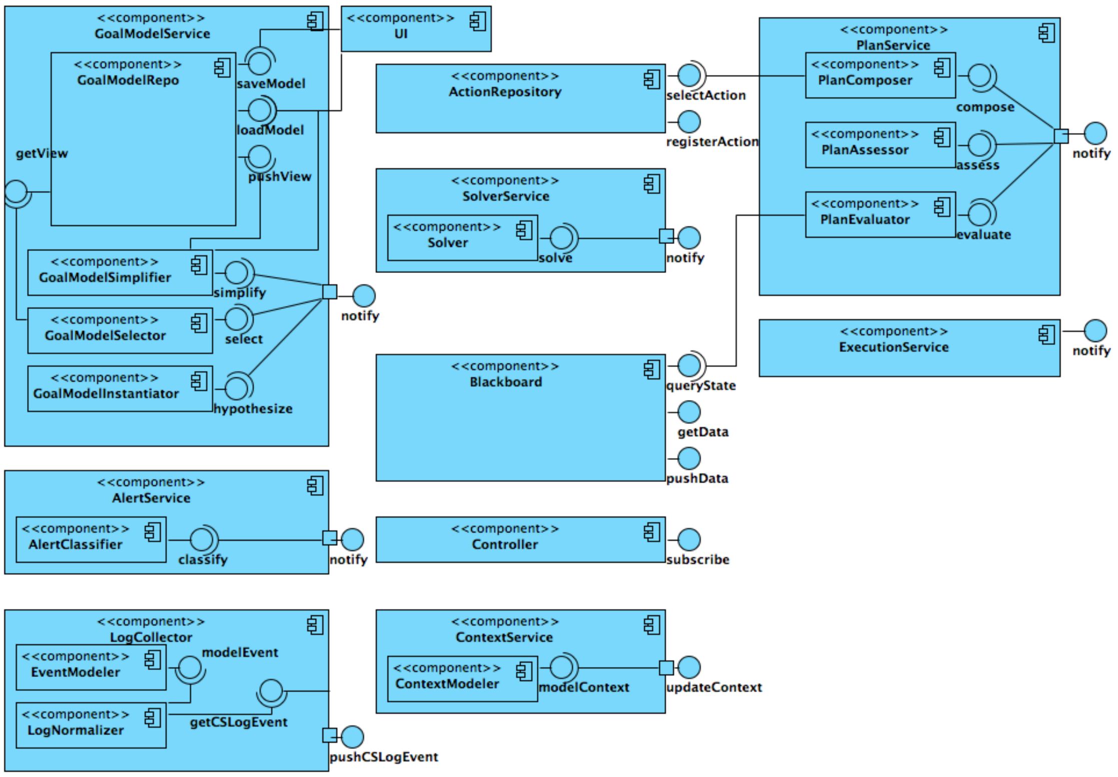
3.1.5 Execute

Finally, the module that changes the behavior of the controlled system is:

***ExecutionService*:** This module closely interacts with the controlled system in order to enact the plan proposed by the self-adaptive framework. The Plan proposed by the PlanService is stored as a set of Actions in the Blackboard component, and encapsulates tasks that the controlled system should execute in order to change its state.

3.2 Architecture Description

In this section we present the component diagram of the self-adaptive framework and describe the functionality of each component in greater detail. In figure 3.2 we depict the component diagram of the proposed architecture containing the core components, their subcomponents, their interfaces and the way in which they interact. We proceed by providing more information for each component as well as sequence diagrams of their internal functionality when needed.



<<component>> GoalModelService
<<component>> GoalModelRepo
saveModel
loadModel
pushView
getView
<<component>> GoalModelSimplifier
simplify
<<component>> GoalModelSelector
select
<<component>> GoalModelInstantiator
hypothesize
notify
<<component>> UI
<<component>> ActionRepository
selectAction
registerAction
<<component>> SolverService
<<component>> Solver
solve
notify
<<component>> Blackboard
queryState
getData
pushData
<<component>> Controller
subscribe
<<component>> PlanService
<<component>> PlanComposer
compose
<<component>> PlanAssessor
assess
<<component>> PlanEvaluator
evaluate
notify
<<component>> ExecutionService
notify
<<component>> AlertService
<<component>> AlertClassifier
classify
notify
<<component>> LogCollector
<<component>> EventModeler
modelEvent
<<component>> LogNormalizer
getCSLogEvent
pushCSLogEvent
<<component>> ContextService
<<component>> ContextModeler
modelContext
updateContext

FIGURE 3.2: Architecture Component Diagram

3.2.1 Blackboard Component

This component is responsible for storing and providing to the other components information that contribute towards composing a final reconfiguration plan for the controlled system. The Blackboard component stores the following types of partial solutions to the problem of self-adaptation: *LogEvenet*, *Alert*, *Goal*, *Hypothesis*, *Model*, *Plan*. During the self-adaptation process, this component must be able to provide a holistic view to the other components of the system of the currently available knowledge by projecting its state and its content.

Blackboard exposes the following interfaces:

- **pushData:** Store information of the types previously mentioned
- **getData:** Retrieve information of a specific type
- **queryState:** Provide information related to the data currently stored in Blackboard

3.2.2 Controller Component

This component orchestrates the execution of all the other components of the self-adaptive framework by consulting the data present on the blackboard. By monitoring the latter's state, it determines which component must execute its functionality next, so that the system can get closer to a final self-adaptation plan.

The controller component implements the Observer design pattern. During the initialization of the self-adaptive framework, the experts of the framework subscribe to the controller declaring the types of partial solutions that they require in order to execute their functionality. Once the controller monitors that a state of the the Blackboard is such that will allow the proper execution of an expert, it sends a notification to this component so it can begin its execution.

Controller exposes the following interfaces:

- **subscribe:** Store an expert that should be notified once data of its interest become available on the Blackboard.

3.2.3 LogCollector Component

This component encapsulates all functionality related with populating the Blackboard with logging events that have been issued from the controlled system. It runs in a separate process from the other components in order to keep pushing log data to the Blackboard, even if an adaptation loop has started.

The controlled system pushes its log data (CSLogEvents) to the LogCollector component, where they are temporarily stored. The **LogNormalizer** subcomponent runs an event loop and consumes all the CSLogEvents that are currently stored to the LogCollector component. The consumed events are then sent to the **EventModeler** subcomponent which transforms the raw log events of the controlled system, into LogEvents which contain information required for the self-adaptation process and which are understood by the other experts of the system. EventModeler is also responsible for filtering out events that are of no interest to the self-adaptive framework, duplicates, or events containing incomplete information. Finally, the created LogEvents are pushed to the Blackboard for other experts to utilize.

LogCollector exposes the following interfaces:

- **pushCSLogEvent:** This interface is used by the controlled system in order to publish its raw logging data to the self-adaptive framework.
- **getCSLogEvents:** This interface is exposed only to LogCollector's subcomponents, and it provides all the LogEvents that are now stored in LogCollector.
- **modelEvent:** This interface processes the unstructured data present in CSLogEvent and return objects of the type LogEvent that the self-adaptive framework can reason on.

The sequence diagram of figure 3.3 presents the process of collecting logging data from the controlled system and updating the contents of the Blackboard component, and is explained below:

loop: LogCollector always runs in a loop.

1. pushCSLogEvent: The controlled system sends an asynchronous message containing log data. The data are temporarily stored in LogCollector.
2. checkEvents: LogCollector checks the existence of new log data sent from the controlled system.

   opt: The following steps are executed if LogCollector detects new log data from the controlled system.

3. modelEvents: EventModeler processes the raw log data and models them into entities of LogEvent type.

3.1 LogEvents: The modeled data are returned back to LogCollector.

4. pushData: LogCollector updates the contents of Blackboard by adding the newly generated LogEvents.

3.2.4 AlertService Component

This component plays the role of the expert responsible for diagnosing execution time issues and raising Alerts that the self-adaptive system needs to address. AlertService

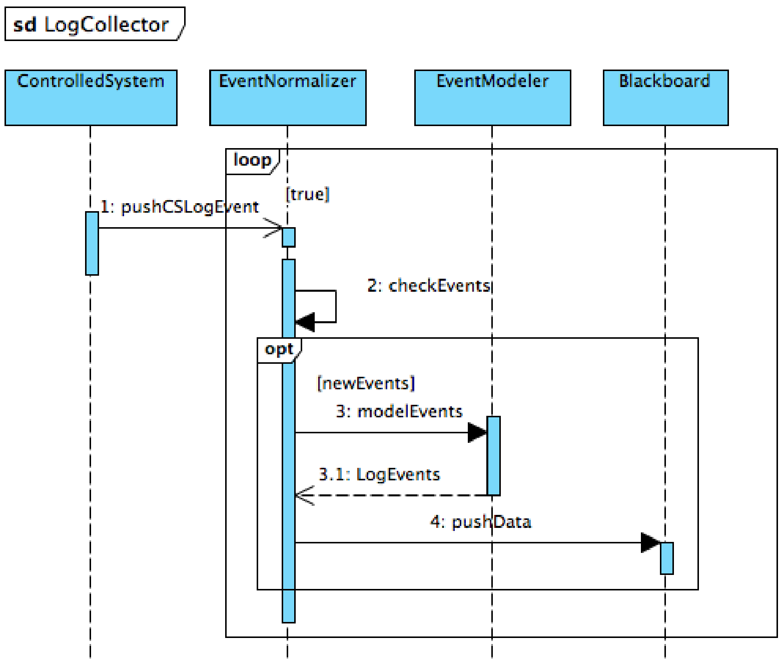


FIGURE 3.3: LogCollector Sequence Diagram

requests LogEvents from the Blackboard component and then provides them to the **AlertClassifier** subcomponent. The latter includes all the required functionality in order to decide whether one or more LogEvents imply that the system has reached an undesirable state and thus an Alert needs to be raised.

AlertService and AlertClassifier implement the Strategy design pattern. Developers can easily alternate the diagnosis part by implementing the AlertClassifier interface and creating their own classifiers which may involve more appropriate diagnosis functions for the domain that the self-adaptive system is applied to. AlertService can also choose different classification strategies during runtime as a result of changing policies of the self-adaptive system.

AlertService exposes the following interfaces:

- **notify:** This interface is implemented by the Controller component in order to inform the AlertService component that Blackboard includes LogEvents that could potentially cause the upraisal of an Alert.
- **classify:** This interface is used internally in the AlertService component to invoke the alert generation process included in the AlertClassifier subcomponent.

The process of raising Alerts is shown in the sequence diagram of figure 3.4 and is explained below:

1. notify: AlertService is notified by the Controller component that LogEvents have become available to the self-adaptive system.

1.1 getData: AlertService requests from Blackboard the LogEvents.

1.2 pushData: LogEvents are sent to AlertService.

1.3 classify: AlertClassifier receives the LogEvents to be classified as possible Alerts.

opt: The following steps are executed if AlertClassifier judges that the monitored events impose a threat to the controlled system and thus an Alert needs to be raised.

1.4 Alert: The generated Alert is sent back to AlertService.

1.5 pushData: AlertService publishes the raised Alert to the Blackboard.

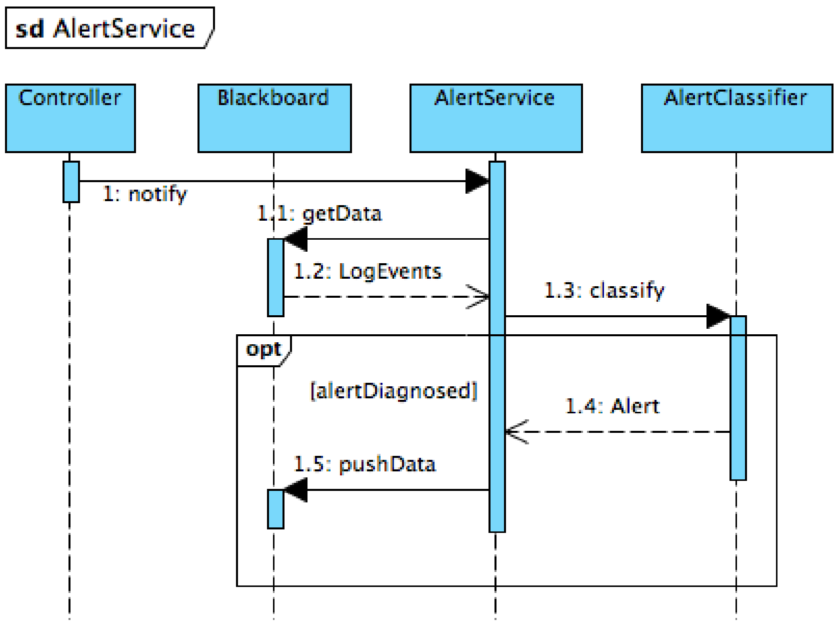


FIGURE 3.4: AlertService Sequence Diagram

3.2.5 GoalModelService Component

This component is responsible for managing all functionality related to Goal Models. Goal Models play a key role in the self-adaptive system by modeling solutions to runtime issues of the controlled system as AND/OR trees and revealing how satisfying a goal can affect another objective of the system. Goal Models are also able to express environment variability by containing contextual constraints that alternate the structure of the Goal Model.

The **GoalModelRepo** subcomponent acts as the storage unit of the defined Goal Model that regulate the adaptation actions of the self-adaptive system. Once an administrator has defined a GoalModel by using the UI component, it stores it in the GoalModelRepo component for the self-adaptive system to use. During the initialization of the system, the self-adaptive system loads the stored Goal Model and makes it available to the GoalModelService. It also stores views of the Goal Model due to applying contextual constraint by the GoalModelSimplifier and makes them available to experts who need them.

The **GoalModelSimplifier** subcomponent applies the contextual constraints that the Blackboard contains to the Goal Model in order to extract a simplified view of the latter. This process is executed every time that a new session of the self-adaptive framework is initiated in order to avoid executing the extraction process (which is run on the whole Goal Model) every time that the adaptation loop is executed.

The **GoalModelSelector** subcomponent is responsible for associating the Alerts generated from AlertService with parts of the defined Goal Model that represent how to deal with the issues that the Alerts represent. Essentially this component defines in which way the controlled system's state should be alternated in order to reach a desirable state. The GoalModelSelector runs when an Alert is published to the Blackboard, and after selecting the appropriate Goals that need to be satisfied for the self-adaptation to be successful, it sends them to the Blackboard so that another expert can reason on them. As the selection process is of paramount importance and highly depends on the nature of the controlled system, it implements the Strategy design pattern so that it can be easily adjusted to user's needs.

The **GoalModelInstantiator** subcomponent serves the purpose of creating a Hypothesis from the selected Goal Nodes. A Hypothesis contains all the nodes of the Goal Model that need to be satisfied so that a re-mediation plan can be devised. The hypothesis generation algorithm is described in detail in Chapter 4. Once the Hypothesis is generated it is then published to the Blackboard.

GoalModelService exposes the following interfaces:

- **notify:** Equivalent to the AlertService, this interface is implemented by the Controller component in order to notify the GoalModelService that data that can trigger its encapsulated functionality has become available to the Blackboard.

- **simplify:** This interface is exposed by the GoalModelSimplifier component and is implemented internally by AlertService during the system's initialization. AlertService is notified that there are Constraints present in the Blackboard from which a view of the Goal Model can be extracted and then invokes simplify for Alert-Simplifier to t

- **select:** This interface is used internally by the GoalModelService component to initiate the Goal selection process which is encapsulated in the GoalModelSelector component.

- **hypothesize:** This interface is used internally by the GoalModelService component to invoke the Hypothesis generation algorithm implemented in the GoalModelInstantiator component.

Figure 3.5 shows the sequence diagram of Goal selection and Hypothesis generation by the GoalModelService and is explained below.

1. notify: The controller notifies GoalModelService that an Alert has been issued.

1.1 getData: GoalModelService requests the Alert from Blackboard.

1.2 Alert: The Alert is returned to the GoalModelService.

1.3 select: GoalModelSelector begins the selection process.

1.3.1 getView: GoalModelSelector request the Goal Model view from the GoalModelRepo component.

1.3.2 ContextualGoalModel: GoalModelRepo replies with the view of the Goal Model.

1.3.3 select: GoalModelSelector executes the Goal selection functionality.

1.3.4 Goals: The selected Goals are sent to the GoalModelService.

1.4 pushData: GoalModelService publishes the selected Goals to the Blackboard

2. notify: The controller notifies GoalModelService that Goals are available for it to hypothesize on.

2.1 getData: GoalModelService request the Goals from Blackboard.

2.2 Goals: The Goals are returned to the GoalModelService.

2.3 hypothesize: GoalModelInstantiator is invokes to generate a Hypothesis using the previously obtained Goals.

2.3.1 instantiate: GoalModelInstantiator runs the instantiation algorithm in order to generate a proper Hypothesis.

2.3.2 Hypothesis: GoalModelService receives the Hypothesis from GoalModelInstantiator.

2.4 pushData: GoalModelService pushed the Hypothesis to the Blackboard.

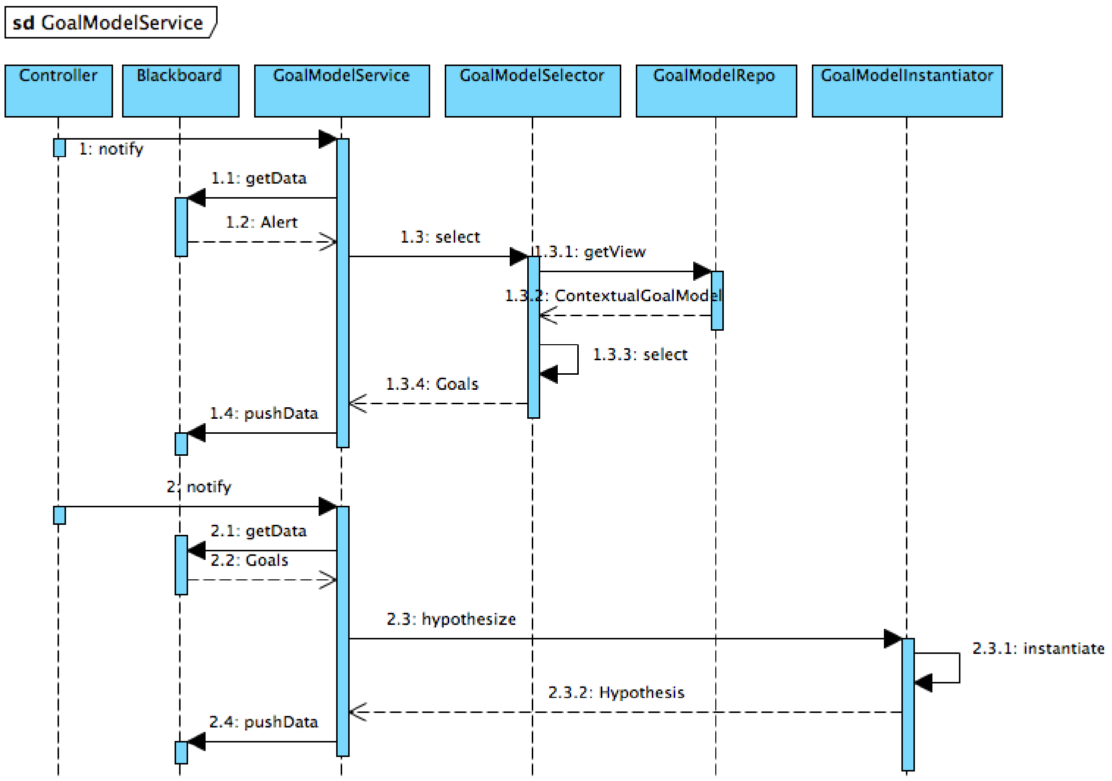


Figure 3.5: GoalModelService Sequence Diagram

3.2.6 SolverService Component

This component is responsible for the validation of the Hypothesis proposed by the GoalModelService. It does so my employing solvers who are able to reason on the Hypothesis stored in the Blackboard.

The **Solver** subcomponent receives the Hypothesis present in the Blackboard and then creates a formula from the Hypothesis. The formula generation process is further explained in Chapter 4. Next, it employs an algorithm to provide a solution for the constructed formula or decide about its unsatisfiability. The Solver subcomponent implements the strategy design pattern so that different solving algorithms can be attached to the framework according to the user's needs.

SolverService exposes the following interfaces:

- **notify:** This interface is implemented by the Controller component in order to inform the SolverService component that another expert has provided a Hypothesis that needs to be checked.
- **solve:** This interface is used internally in the SolverService subcomponent and is mandatory for all solvers to implement. SolverService invokes it in order to retrieve a model from the Hypothesis, if the latter is satisfiable.

Figure 3.6 depicts the sequence diagram related to the extraction of satisfying models out of a Hypothesis.

1. notify: The controller notifies SolverService that a Hypothesis is available to be check for satisfiability.

1.1 getData: SolverService requests the Hypothesis from Blackboard.

1.2 Hypothesis: The Hypothesis is sent to SolverService.

1.3 solve: Solver begins to reason on the Hypothesis.

1.3.1 generateFormula: Solver creates a formula from the provided Hypothesis.

opt: The following steps are only executed if the formula is proven to be satisfiable by the Solver.

1.3.2 getModel: The Solver is asked to provide the model that satisfies the formula.

1.3.4 Model: The provided model is sent back to the SolverService.

1.4 pushData: SolverService publishes the model that satisfies the Hypothesis to the Blackboard.

3.2.7 PlanService Component

This component is responsible for compiling the Model that satisfies the Hypothesis into a reconfiguration plan. Afterwards, the self-adaptive system needs to actuate the plan's Actions on the controlled system in order to change its state. **PlanService** interact with the Blackboard to provide to its subcomponent the Model on which the Plan is based as well as runtime parameters that are required for evaluating the Plan prior to its execution.

The **PlanComposer** subcomponent fulfills the objective of transforming a Model into a Plan. It does so, by iterating over the Model's Tasks and selecting the Actions from

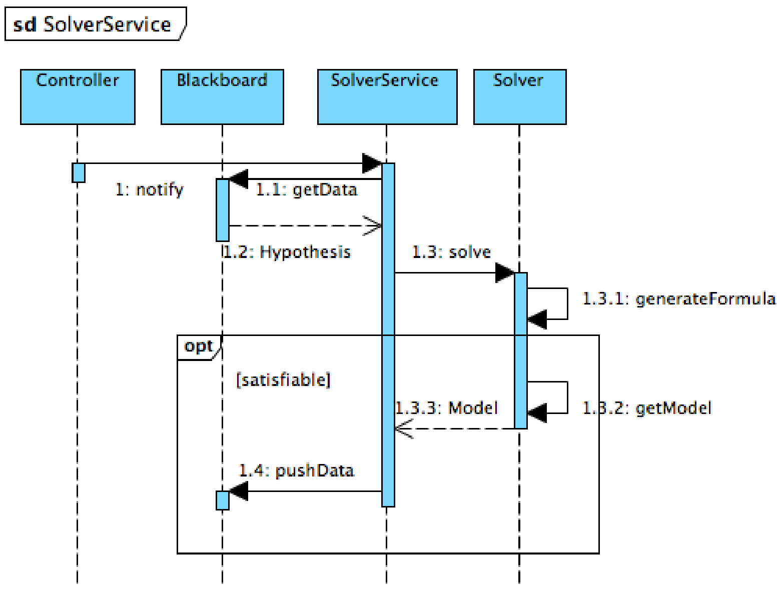


FIGURE 3.6: SolverService Sequence Diagram

the **ActionRepository** component that corresponds to each Task. Actions are the building blocks of an execution Plan and are bijective to Tasks. Tasks are part of the Goal Model and are discussed in Chapter 4.

The **PlanAssessor** subcomponent contributes to the process of creating an execution Plan only when more than one Models have been provided to the PlanService. This scenario arises when the **Solver** utilized is capable of providing more than one Models that can satisfy the Hypothesis. PlanAssessor is a subcomponent whose functionality must be defined by the user and thus it implements the Strategy design pattern.

The **PlanEvaluator** subcomponent returns the final verdict regarding whether each Action of the composed Plan should be executed or not, according the current state of the controlled system. In order to reach such a decision, PlanEvaluator queries the Blackboard in order to observe current parameters of the running system, which are included in the LogEvents entities. PlanEvaluator finally produces a Plan containing only the Actions that should be executed. For the same reasons with PlanAssessor, PlanEvaluator implements the strategy design pattern.

PlanService exposes the following interfaces:

- **notify:** This interface is implemented by the Controller component in order to notify the PlanService that SolverService has provided models that satisfy the proposed Hypothesis.

- **compose:** This interface is exposed by the PlanComposer subcomponent and is implemented internally by PlanService. It includes the required functionality for the compilation of a Model into an executable plan.
- **assess:** This interface is used by the PlanService component when more than one Plans have been created by the PlanComposer subcomponent. It returns ranked Plans according to the assessing criterion of the PlanAssessor component.
- **evaluate:** This interface is implemented by the PlanService for it to decide whether the Actions that constitute the composed Plan are still appropriate for execution under the current state of the controlled system.

Figure 3.7 depicts the sequence diagram related to the compilation of a Model into a final execution Plan.

1 notify: The controller notifies PlanService that a Model is ready to be processed and compiled into an execution Plan.

1.1 getData: PlanService requests the Model from Blackboard.

1.2 Model: The Model is sent to PlanService.

1.3 compose: PlanComposer initiates the Plan compilation process.

loop: Steps 1.3.1, 1.3.2, 1.3.3 are executed for each Task in the Model.

1.3.1 getAction: PlanComposer requests from ActionRepo the Action corresponding to the current Task.

1.3.2 Action: The Action is sent back to PlanComposer.

1.3.3 addActionToPlan: The Action is added to the Plan.

1.3.4 Plan: The Plan is returned to the PlanService.

opt: Steps 1.4, 1.5 are executed only if more than one Plans have been composed, due to the existence of more than one satisfying Models.

1.4 assess: PlanAssesor executes its ranking algorithm asses all proposed Plans.

1.5 AssessedPlan: The AssessedPlan is returned to the PlanService.

1.6 evaluate: PlanEvaluator begins to evaluate the feasibility of the proposed Plan.

1.6.1 queryState: PlanEvaluator requests runtime parameters of the controlled system that are stored into Blackboard.

1.6.2 RuntimeInformation: The RuntimeInformation are sent back to PlanEvaluator.

loop: The following step is executed for each Action of the composed Plan.

1.6.3 isFeasible: The Actions feasibility is evaluated according to the Runtime information. If the verdict is *feasible* the Action is included in the final execution Plan.

1.6.4 Plan: The final execution Plan is returned to PlanService.

1.7 pushData: The final execution Plan is pushed to the Blackboard.

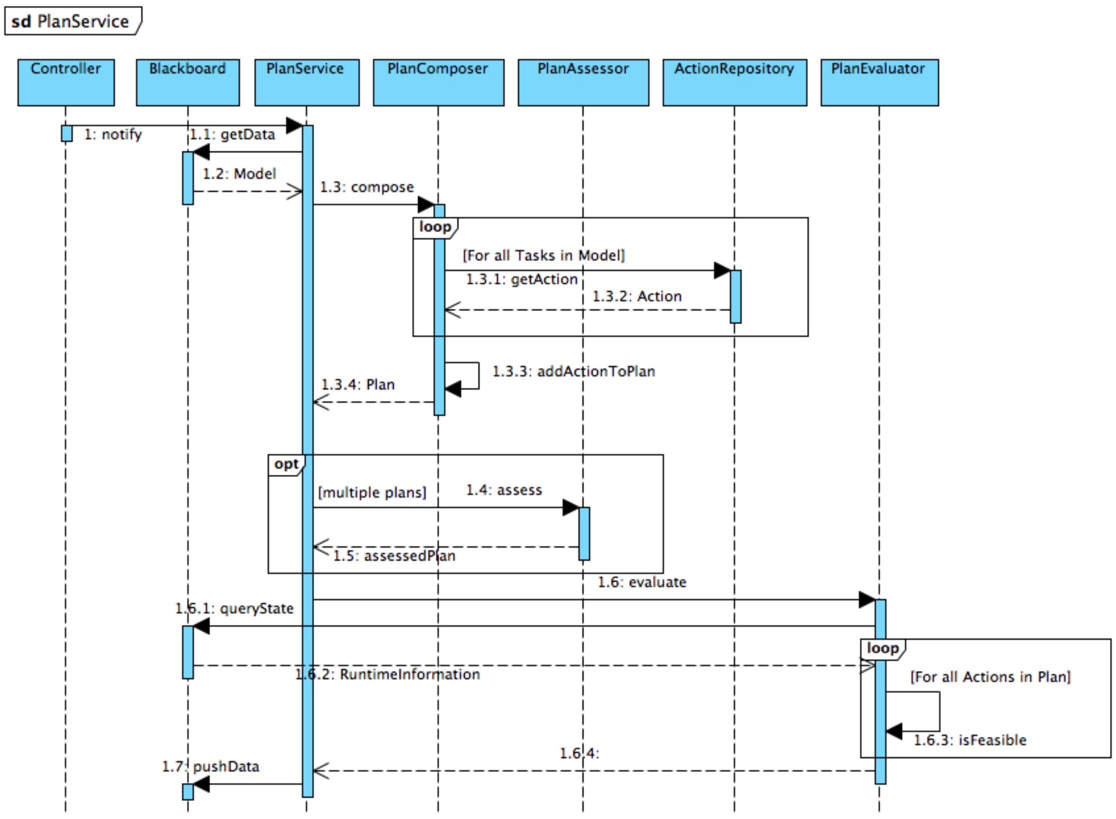


FIGURE 3.7: PlanService Sequence Diagram

3.2.8 ExecutionService Component

This component is run last in the adaptation loop and acts as an effector to the controlled system. Once notified by the controller for the existence of an execution Plan, it receives it from the Blackboard and proceeds by executing one-by-one the Actions it contains.

3.2.9 ActionRepository Component

This component serves as the storage unit of Actions. In our self-adaptive framework, Actions play the role of *touchpoints*. Actions include low-level functionality of the controlled system and must be implemented by the developer.

3.2.10 UI Component

This component enables the users to define their solutions to possible runtime issues of the controlled system as Goal Models. The user can create multiple Goal Trees through a graphical user interface, link goals with contributions, declare costs or benefits of each goal and add contextual constraints to their solution. Once the Goal Model is created, it is then stored in the GoalModelRepo subcomponent and is utilized by the self-adaptive framework during its Analyze phase.

3.2.11 ContextService Component

This component is responsible for introducing Constraints relevant to the controlled system's environment to the self-adaptive system. These Constraints are used by the GoalModelSimplifier subcomponent in order to create a view of the Goal Model that the users of the self-adaptive system have defined. This ContextualGoalModel represents how the environment of the controlled system (i.e. the policies applied to it) influence the decision making process modeled in the Goal Model.

The **ContextModeler** subcomponent interacts with the controlled system in order to receive information relevant to its environment and policies. Once the information is retrieved, ContextModeler is responsible for modeling them into Constraints so that the **GoalModelSimplifier** subcomponent can extract a view of the user-defined Goal Model that conforms to the applying Constraints.

3.3 Component Interaction and Functionality of the Framework

Having seen in detail the functionality of each component of the self-adaptive system we now present the main execution of the self-adaptive loop by showing the interaction between its basic components. Figure 3.8 depicts the sequence diagram of the self-adaptive control loop explaining the order in which the multiple experts of the system contribute towards devising and executing a Plan that will change the controlled system's state.

1 UpdateContext: At the beginning of the adaptation session, the controller requires that the ContextService inserts any contextual Constraints of the controlled system to the Blackboard.

1.1 pushData(Constraints): ContextService published the modeled Constraints to the Blackboard.

2 adapt: Controller enters the self-adaptation loop

loop: The following steps are executed as long as the controlled system is running.

3 queryState: Controller queries the Blackboard to check for new LogEvents that the LogCollector might have provided.

4 notify-classify: Controller informs the AlertService that it can execute its classification process with LogEvents available from the Blackboard.

4.1 pushData(Alert): AlertService publishes the diagnosed Alert to the Blackboard.

5 notify-select: Controller informs the GoalModelService that it can execute its Goal selection strategy.

5.1 pushData(Goals): GoalModelService returns the selected Goals that should be satisfied for the system to remediate the monitored Alert.

6 notify-hypothesize: Controller informs the GoalModelService that it can execute its hypothesis generation algorithm based on the previously selected Goals.

6.1 pushData(Hypothesis): GoalModelService pushes the generated Hypothesis to the Blackboard.

7 notify-solve: Controller informs the SolverService that a Hypothesis has been issued and it can be checked for satisfiability.

7.1 pushData(Model): ModelService returns the satisfying instance of the Model if the Hypothesis can hold, otherwise it returns unsatisfiable.

opt: The following steps are executed only if the Hypothesis is proven to be satisfiable and a Model for it is provided.

8 notify-compose: Controller informs the PlanService that one or more Models that satisfy the Hypothesis are present in the Blackboard.

8.1 pushData(Plan): PlanService returns the final execution Plan containing only the Action that the controlled system needs to execute in order to remediate the diagnosed abnormality.

9 execute: Controller notifies the ExecutionService that a Plan has become available to execute on the controlled system.

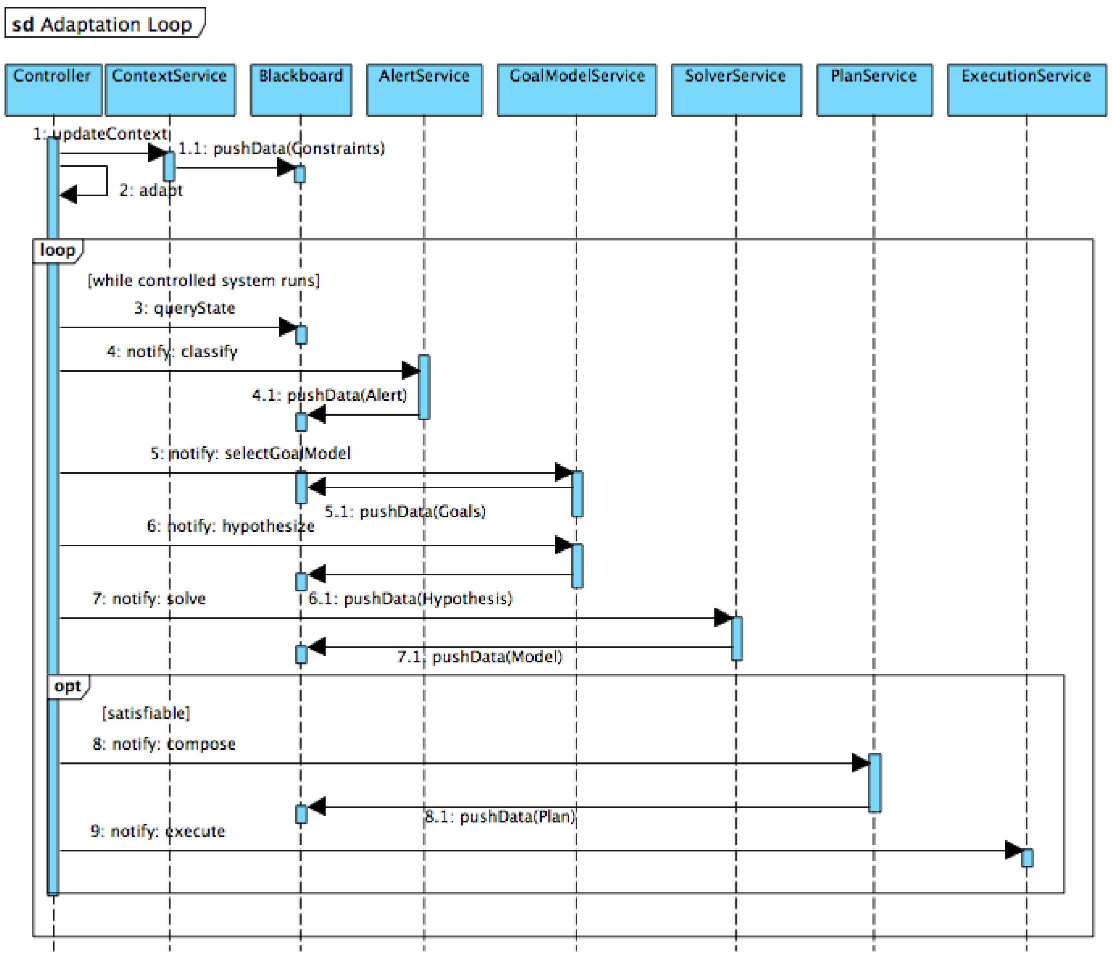


Figure 3.8: Self-adaptive Framework Sequence Diagram

Chapter 4

Domain Models

In this chapter we discuss the Domain Models of our prototype implementation. We begin by providing the Domain Model of the Task Resolution Model specification, and then we visit the Domain Model of the core entities of the self-adaptive system. Finally we present the Domain Model of the extensions we applied to the CloudSim framework.

4.1 Task Resolution Model Specification

Task Resolution Models, which are formally defined in chapter 5, constitute the basis of our self-adaptive system as they are responsible for modeling the actions that the IT experts, system administrators and managers wish to be triggered by the self-adaptive system and executed by the controller system, so that the monitored problems of the latter can be re-mediated.

By utilizing our proposed models, IT professionals can describe solutions to specific Alerts in the form of Goal Trees. As explained in section 2.2, Goal Trees are graphs of goals which can be AND/OR decomposed to other goals expressing that in order to achieve a goal one should also achieve the goals it is decomposed to. Through these decompositions, goals are eventually broken down to concrete, atomic, self-contained tasks that the controlled system needs to execute.

Our Task Resolution Model specification differentiates from the traditional Goal Models by i) allowing multiple Goal Trees as part of the Goal Model; ii) associating each Goal and Task with a cost/benefit value; iii) including contextual elements which according to the controlled system's execution environment, can omit some parts of the model for this session.

The tight relation between users and Task Resolution Models as well as their pivotal role in the self adaptive framework, constitute them as a part of the framework that needs to be robust, easily extensible and well understood by its users. In order to achieve this, their Domain Model follows the MOF specification[31] and is implemented using the Eclipse Modeling Framework[32].

Figure 4.1 depicts the Task Resolution Model Domain Model in the ecore meta-modeling language provided by EMF. We proceed by describing the classes and interfaces of the proposed Domain Model.

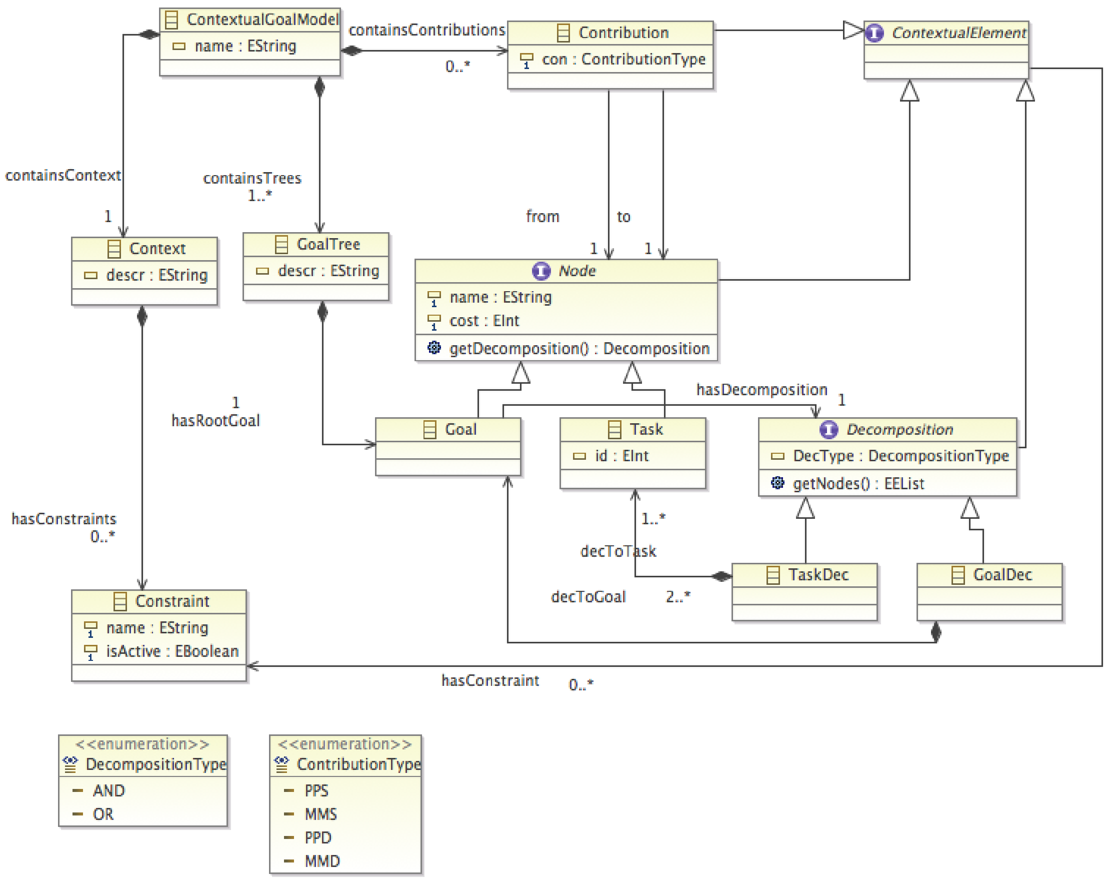


FIGURE 4.1: Task Resolution Model Domain Model

4.1.1 Class ContextualGoalModel

This class serves as the container of the proposed model. It has the single attribute `name` of type `String`. It is related to the Contribution class through the *containsContributions* composition relation with a `0 ... *` multiplicity. It has the composition relation *containsContext* of multiplicity `1` with the class Context and the composition relation *containsTree* with the class GoalTree of multiplicity `1 ... *`. These containment relations show the three basic elements that our Goal Model is composed by.

4.1.2 Class GoalTree

This Class represents each Goal Tree that is included in the Task Resolution Model. Goal Trees model a solution modeled by an IT expert as an AND/OR tree. The class has the attribute `descr` of type `String` that describes the solution that this Goal Tree aims to provide. It is related to the class ContextualGoalModel as described above, and with the class Goal through the *hasRootGoal* composition relation of multiplicity `1`.

4.1.3 Interface Node

This interface defines the behavior and the attributes that each node of the Task Resolution Model graph must include. The attribute `name` of type `String` declares the name of the Node while the attribute `cost` of type `int` showcases the cost associated with the satisfaction and achievement of this Node. All the classes that implement the Node interface must include the operation `getDecomposition` of type `Decomposition` which returns the Decomposition Class containing all the nodes in which the father node is decomposed to. Finally, the Node interface implements the ContextualElement interface, meaning that if certain rules apply, a Node might not be visible in the Task Resolution Model.

4.1.4 Class Goal

This Class implements the Node interface and thus contains all its attributes and defined operations. It represents the concept of a goal that the self-adaptive system wishes to achieve and is the building block of our model. Goals are high-level descriptions of solutions to specific issues related with the controlled system, and when connected via decompositions or contributions, constitute a strategy that the controlled system must follow in order to reach the desirable state. A Goal can be decomposed either to one or more Goals or to one or more Tasks. The goals in which the original Goal is decomposed to, aim to refine the way in which the goal represented by their ancestor can be eventually met. Finally the Goal Class has a `hasDecomposition` association of multiplicity `1` with the Decomposition Interface (which is explained in a next section) making it mandatory that a Node of type Goal is decomposed to other Nodes.

4.1.5 Class Task

This Class also implements the Node interface, containing all its attributes and operations and it represents a task that the self-adaptive system proposes to the controlled system for execution. Consequently, this task is associated with a respective Action from the side of the control system and is described in terms of the latter. The Task class requires the extra parameter `id` of type `int` playing the role of a unique identifier, needed for the selection of the respective Action that the controlled system must execute. Contrary to the Goal class, this class is not associated with the Decomposition interface, meaning that a Task element can be no longer decomposed to other, simple Nodes of the Task Resolution Model. This modeling decision along with the fact that each Goal Tree has a unique root goal, guarantee that our model does not contain loops of decompositions.

4.1.6 Interface Decomposition

This interface declares the attributes and operations that a Decomposition of the Task Resolution Model must include. The attribute `DecType` of type `DecompositionType` (which is explained in a next section) declares whether the class represents an AND or an OR decomposition. The operation `getNode` of type `List` returns all the objects that implement the Node interface and are included in the decomposition. Finally, it implements the ContextualElement interface so that an object that implement the Decomposition interface can be omitted from the model if some conditions are satisfied.

4.1.7 Class GoalDec

This class implements the Decomposition interface and thus it must have a `Dectype` attribute as well as a `getNodes` method. It is compositionally related with the Goal class through the *decToTask* relation which has a `2 ... *` multiplicity. In this way, this container class makes clear that a Goal Node can be AND/OR to at least two subgoals.

4.1.8 Class TaskDec

This class also implements the Decomposition interface and thus it must have a `Dectype` attribute as well as a `getNodes` method. Similarly to GoalDec, it is compositionally related with the Task class through the *decToTask* relation which has a `1 ... *` multiplicity. Conceptually, this class reassures that a Goal that has a Decomposition of the instance TaskDec can be decomposed to one or more Tasks.

4.1.9 Class DecompositionType

This class defines the two different Decomposition types between Nodes as an enumeration. The enumeration contains the values AND and OR.

4.1.10 Class Contribution

This class represents the Contribution entity of the Task Resolution Model. Contributions model the potential side-effect of satisfaction or denial of a Node of the model. It contains the `conType` attribute of type `ContributionType`, which is analyzed later, and has two associations with the Node interface. The *from* and *to* associations refer to the source and the target Node of the contribution respectively, as explained in section 2.2.

4.1.11 Class ContributionType

This class defines the different Contribution types between two Nodes of the proposed model as an enumeration. The values contained are i) PPS representing the `++S` contribution; ii) PPD representing the `++D` contribution; iii) MMS representing the `--S` contribution; iv) MMD representing the `--D` contribution.

4.1.12 Class Context

This class serves as the container of Constraints that arise from the controlled system's execution environment. It includes the attribute `descr` of type `String` which serves as a description of what can be contained in the Context Class. It is compositionally related to the Constraint class (which is explained next) through the *hasConstraints* relation of multiplicity `0 ... *`.

4.1.13 Class Constraint

This class plays the role of Constraints as explained in section 2.2. It contains the attribute `name` of type `String` indicating the name of the Constraint and the attribute `isActive` of type `Boolean`, which indicates whether this Constraint is active through an execution session of the controlled system.

4.1.14 Interface ContextualElement

This interface aims to identify these elements of the Task Resolution Model on which Contextual Constraints can potentially apply. Thus, all classes and interfaces that implement it, are related to the Constraint class through the *hasConstraint* association of multiplicity `0 ... *`.

4.2 Self-Adaptive System

In this section we present the core entities affiliated with the self-adaptive system and the relations between them. We also depict how the internal entities of the self-adaptive framework interact with those that belong to the controlled system and what infrastructure the framework provides to developers in order to assist them in the communication of the two systems.

Figure 4.2 showcases the domain model the self-adaptive system. We proceed by visiting each class and interface of the system.

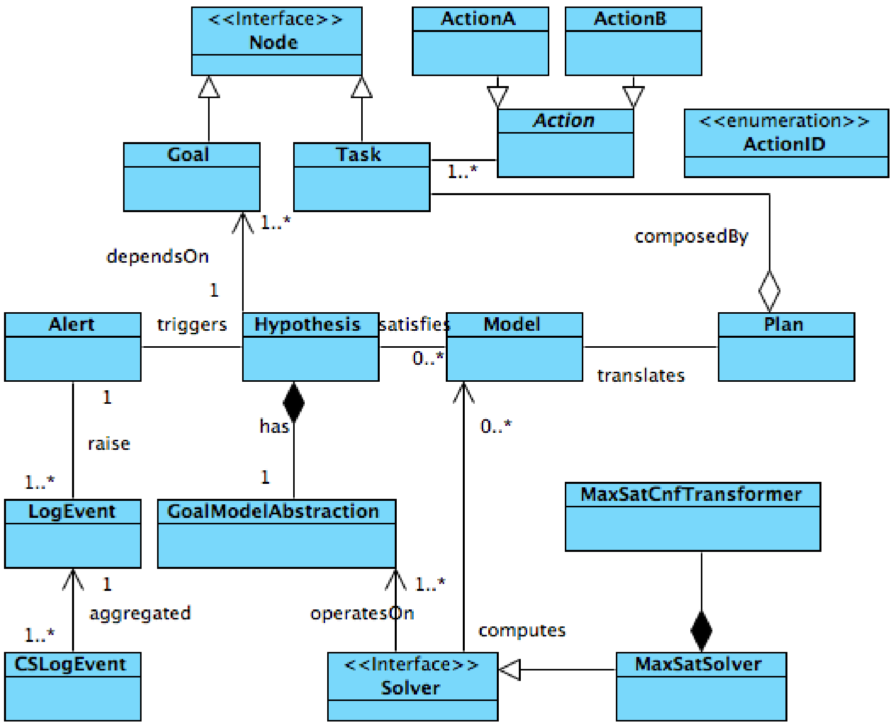


FIGURE 4.2: Self-Adaptive System Domain Model

4.2.1 Class CSLogEvent

This class represents logging data that the controlled system generates and then channels to the self-adaptive system. It is related to the LogEvent class (which is presented next) through the one-way aggregated association of multiplicity 1 . . . *. The log data that this class carries are stored in JSON format.

4.2.2 Class LogEvent

This class represents normalized log data that the self-adaptive system is in position to interpret. One or more CSLogEvents constitute towards the creation of a LogEvent. This Class is related with the Alert class through the raises association of multiplicity 1 . . . *. The AlertService component after examining and correlating LogEvents might generate an Alert that is caused by a set of LogEvents, showcasing this relation.

4.2.3 Class Alert

This class models an issue that was diagnosed by the self-adaptive system. It bears with it information related to why it was classified as an alert as well as the the set of LogEvents that caused the upraisal of the Alert. It is related to the Hypothesis class through the triggers mutual relation of multiplicity 1.

4.2.4 Class Hypothesis

This class includes the Goals for the satisfaction of which, the self-adaptive system seeks execution plans that will alter the controlled system's state. Therefore it is associated with the Goal class described in section 4.1.4 through the dependsOn relation of multiplicity 1 . . . *. In case a Hypothesis is proven to be satisfiable, it is associated with one or more models that make it valid, as the satisfies relation of multiplicity 0 . . . * shows. Finally, the Hypothesis class is compositionally associated with the class GoalModelAbstraction which is analyzed next.

4.2.5 Class GoalModelAbstraction

This class is responsible for abstracting the structure of the Task Resolution Model that the self-adaptive system employs to an abstract representation of Goal Models. It provides an interface to add the elements of the model that need to be satisfied or denied, as well as other key information such as the cost or benefit of a Goal. After the Goals of the Goal Model that the Hypothesis needs to satisfy are added to the abstraction along with their subgoals and their contributions, the GoalModelAbstraction class generates the AND/OR rules from the added elements of the model (as explained in chapter 5) and stores them in a format that Solvers can interpret. Therefore the GoalModelAbstraction class serves as a specification of the Task Resolution Model that is common amongst all solvers that the self-adaptive system utilizes.

4.2.6 Class Model

This class represents the Models containing assignments to the Goals and Tasks of the Task Resolution Model that satisfy the given Hypothesis. Models are provided from Solvers and have a satisfies relation with each instance of the Hypothesis class of multiplicity 0 . . . 1.

4.2.7 Interface Solver

This interface must be implemented by each Solver or reasoning entity that the self-adaptive system wishes to employ. Each Solver is associated with the Models it computes through the computes relation of multiplicity 0 . . . *. The existence of the Solver interface makes changing between different types of Solvers during the system's execution easier and more robust by implementing the strategy design pattern. Also, a Solver is associated with the GoalModelAbstraction class with the operatesOn relation of multiplicity 1 . . . *.

4.2.8 Class MaxSatSolver

This class implements the Solver interface previously described and is responsible for checking a formula for max-satisfiability. It is also related to the MaxSatCnfTransformer through a composition relation in order to be able to transform the given GoalModelAbstraction to a cnf formula.

4.2.9 Class MaxSatCnfTransformer

This class includes all the functionality required to parse the information stored in the GoalModelAbstraction class in oder to transform it to a cnf formula. The MaxSatSolver can then accept this formula as input and check its satisfiability.

4.2.10 Class Plan

This class represents the execution Plan that the self-adaptive system composes for the controlled system to execute. Each instance of the Plan class is associated to the Model class through the `translate` relation. Each Plan is composed by multiple instances of the Task class, a fact indicated by the aggregation relation between the two, of multiplicity `1` ... `*`.

4.2.11 Class Action

This abstract class represents an atomic set of operations executed by the controlled system. It therefore contains operations that depend on the application domain of the controlled system and acts as an effector to the latter. This abstract class defines the parts of the operations that are necessary for each Action, i.e. preparing the message that will be sent to the controlled system.Developers of self-adaptive systems need to extend this class of the framework in order to define their concrete, atomic actions that will affect the controlled system's state.

Each instance of a concrete implementation of the abstract class Action is associated with a single Task of the Task Resolution Model. Finally, this abstract class includes two abstract methods that each concrete action must implement; *isFeasible* and *execute.* The first is responsible for determining wether an Action can be executed according to the current state of the controlled system and it does so by interacting with the latter. The second is responsible for the actual execution of the action on the controlled system.

4.2.12 Class ActionID

This class serves as an enumeration of the different Actions that the developers of the self-adaptive system have implemented. Actions are identified by their unique identifiers which are used in order to execute the corresponding Action of a Task while executing the devised Plan.

4.3 CloudSim Extensions

In this section we present the required extensions that were introduced to the CloudSim modeling framework. The main objective of these additions is to improve the reporting mechanism of CloudSim's core entities so that the implemented self-adaptive system has sufficient information to reason on.

In order to produce the required logging data of a controlled system, developers need to identify the latter's entities which must be managed by the self-adaptive system. In the case of the CloudSim framework, these managed resources are Hosts, Virtual Machines and Cloudlets. The Host class of CloudSim represents a physical host of the data-center, the VM class represents virtual machines provisioned on the physical hosts, and the Cloudlet class serves as an abstraction of applications that are executed on provisioned virtual machines.

The metrics of these core entities are utilized in order to define key performance indicators (KPI) for the managed IT infrastructure. Since these KPIs are related to IaaS, they are grouped into the four main resource types that IaaS can provide: CPU, Memory, Bandwidth and Storage. In Appendix A the metrics generated by our extended version of CloudSim as well as the Key Performance Indicators defined based on them, are listed.

Figure 4.3 shows the domain model of the implemented changes.

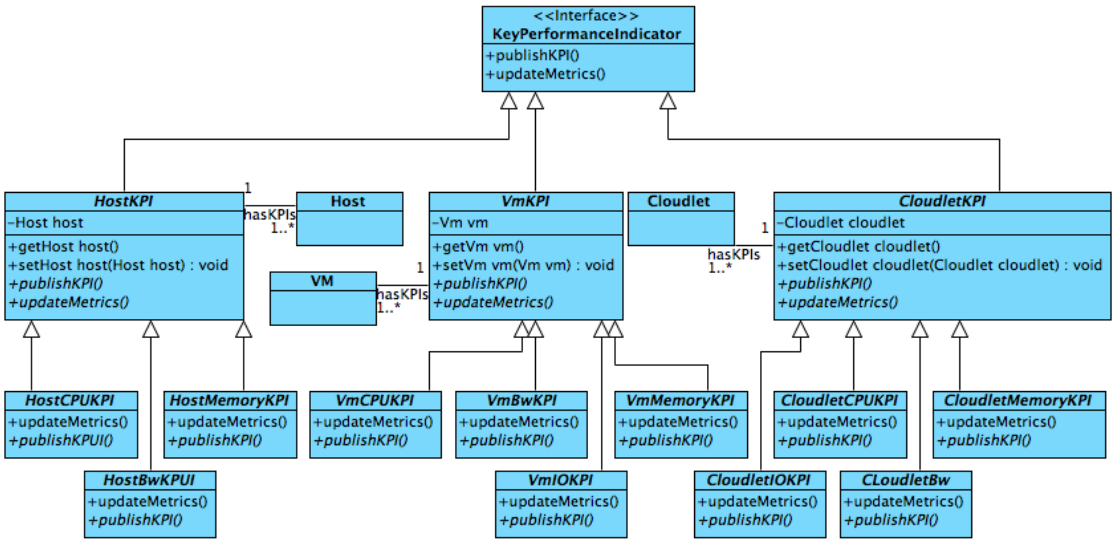


Figure 4.3: Key Performance Indicator Domain Model

Before we explain each class and interface of the domain model, we discuss the design decisions taken into account for their definition. Our main consideration was to relieve developers of the self-adaptive system from extracting the required metrics from the CloudSim framework. For this reason all the functionality of gathering and calculating this information is abstracted from the final user of the CloudSim framework and is implemented as operations of abstract classes defined in our Domain Model. The only obligation of the developer is to extend these classes and then define and calculate the desired KPIs by utilizing the gathered metrics.

4.3.1 Interface KeyPerformanceIndicator

This interface declares the main operations that a KPI object must implement. These are the **updateMetrics()** and **publishKPI()** methods. UpdateMetrics is responsible for gathering the required information for each KPI and publishKPI for calculating the defined KPI from the available metrics and then making it available to the self-adaptive system.

4.3.2 Class HostKPI

This abstract class implements the KeyPerformanceIndicator interface and includes the functionality required by all types of KPIs that can be defined for the Host entity. It is associated with a single Host element in order to contain information only related to that specific Host. An example of an operation that each KPI related to the Host entity must provide is creating log activity to a specific information channel.

This abstract class is extended by the abstract classes **HostCPUKPI**, **HostMemoryKPI** and **HostBwKPI**. Each of them implements the updateMetrics() operation in order to gather their respective required metrics from the CloudSim framework. All the metrics related to the host entities and for each associated report that our extensions implement are listed in table A.3. Some example of CPU metrics for instance, are *number of cores* on each physical host, *MIPS* (million instructions per second) *per core*, *total requested MIPS per VM* allocated on Host etc.

The obligation of the final user is to extend the lastly mentioned abstract classes (HostCPUKPI etc.) into classes responsible for computing a specific KPI that is related to the resource type of the abstract class that they implement (i.e. memory). For example, to define the KPI *Percentage of Total Mips Utilization,* a developer should extend the HostCPUKPI abstract class which contains the required metrics shown in the left column of table A.3, and then use these metrics in order to calculate and publish the KPI to the self-adaptive system.

4.3.3 Class VmKPI

This abstract class is analogous to the HostKPI abstract class and contains the same functionality as the latter, only now related to the Vm entity of the CloudSim framework. It is associated with CloudSim's Vm class with a multiplicity of 1 so that each KPI related to virtual machines metrics contains information for a specific Vm entity.

The same structure for the different types of KPIs according to the resource types they depend on, is followed in the case of VmKPI mimicking HostKPI. However in the case of virtual machines, an extra resource type needs to be monitored, that being i/o operations. For this reason, VmKPI is extended by the extra abstract class **VmIOKPI**. Examples of the monitored metrics that these abstract classes keep track of are *requested IOPS* (input output operations per second) by a Vm, *allocated BW* to a virtual machine, *requested mips per core* dedicated to a Vm.

The final user needs to implement the same functionality as in the case of the Host element, extending the VmIOKPI, VmCPUKPi, VmMemoryKPI, VmBwKPI abstract classes and next calculating and publishing its desired KPIs to the self-adaptive system.

4.3.4 Class CloudletKPI

This abstract class is the equivalent of the HostKPI and VmKPI abstract classes target towards Cloudlets. Cloudlets are the core entities of the CloudSim framework representing applications that run on provisioned virtual machines, and with this class we aim to define important KPIs that showcase the response of the IT infrastructure to the demands of applications.

CloudletKPI works in an identical way to the VmKPI abstract class. The CloudletCPUKPI, CloudletMemoryKPI etc. abstract classes gather the important metrics presented in A.1. Extending these classes, enables the user to define its desired KPIS such these shown in the previous table. Examples of such KPIs include the *total cpu throughput* of a Cloudlet, the *active average memory throughput* of a Cloudlet etc.

Chapter 5

Reasoning Framework

In this chapter we explain the reasoning mechanism that our self-adaptive system employs and the models that it utilizes. We present the *Task Resolution Model* (whose entities were also described in the previous chapter) which provides to developers and administrators a high level way to define re-mediation plans that should be applied to the controlled system when an undesired state has been reached. We provide an instantiation algorithm for the Task Resolution Model which guarantees that all the parts of the model that can contribute towards compiling a re-mediation plan for the controlled system are included in the reasoning process. Next, we show how the plan compilation problem is reduced to SAT and finally the derivation of the cnf formula that is fed to the SAT solver.

5.1 Task Resolution Modeling

In this section we formally define the modeling framework utilized by the self-adaptive system in order to resolve an execution Plan. Our models are a variation of the model presented in [33].

Definition 1: A *Task Resolution Model* can be formalized as a tuple $TRM = \langle \mathcal{N}, \mathcal{R}, \mathcal{C} \rangle$ where (i) $\mathcal{N} \neq \emptyset$ is a set of *nodes*, (ii) $\mathcal{R} \neq \emptyset$ is a set of *rules* between nodes and (iii) $\mathcal{C}$ is a set of *contextual tags*. Let $\mathcal{G}$ and $\mathcal{T}$ be two disjoint non-empty sets of *goal* and *task nodes*, and $\mathcal{R_D}$ and $\mathcal{R_B}$ two disjoint non-empty sets of *decomposition* and *binary rules* respectively. Then the set of *nodes* is defined as $\mathcal{N} = \mathcal{G} \cup \mathcal{T}$ and the set of *rules* as $\mathcal{R} = \mathcal{R_D} \cup \mathcal{R_B}$. Each *node* and *rule* can be assigned a *contextual tag*.

The above definition explains how nodes of the proposed model are connected to each other through rules. Decomposition rules are n-ary relations that showcase how a node can be AND/OR decomposed to its offsprings. In figure 5.1 we see that for the goal *Provision Resources* to be satisfied, at least one of its offspring *ProvisionNewHostAndMigrateVM* and *IncreaseHostProcessingPower* needs to be satisfied. Binary rules are relations that express the side-effects that source node can impose to the target node. In that sense, satisfaction of goal *Provision Resource* denies the goal *DecreaseEnergyConsumption*. Next we define these two types of relations.

Definition 2: We denote a *decomposition rule* $r_d \in \mathcal{R_D}$ as the tuple $r_d = \langle T_d, p, O \rangle$ where $(i) T_d \in \{AND, OR\}$ is the decomposition type, (ii) $p \in \mathcal{G}$ is the *parent node*, (iii) $O = \{o_1, o_2, \ldots\}$ with $o_1,\ o_2,\ \ldots \in \mathcal{N}$ is the non-empty set of *offspring nodes*. In an equivalent manner, we denote a *binary rule* $r_b \in \mathcal{R_B}$ as the tuple $r_b = \langle T_b, s, t \rangle$ where (i) $T_b \in \{++\ S/D, --\ S/D\}$ is the binary relation type, (ii) $s \in \mathcal{N}$ is the *source node*, (iii) $t \in \mathcal{N}$ is the *target node.*

With the above relations we can now define the notions of *parent* and *source* nodes as following:

Definition 3: Given a *node n*, its parent node is defined as $parent(n) = \{p \in N \mid \exists r_d = \langle T, p, O \rangle \mid n \in O\}$.

Definition 4: Given a *node n*, the set of its source nodes is defined as $S = source(n) = \{s_i \in N \mid \exists r_b = \langle T, n, s_i \rangle \in R_B\}$.

We proceed to define *goal trees*, the building block of the reasoning framework.

Definition 5: The non-empty subsets $G_i \subset \mathcal{G}, T_i \subset \mathcal{T}, R_i \subset \mathcal{R}$ where $N_i = G_i \cup\ T_i$, *is a goal tree* $GT = \langle N_i, R_i \rangle$ *if an only if* G_i contains exactly one *root goal* r_g : $parent(r_g) = \emptyset$.

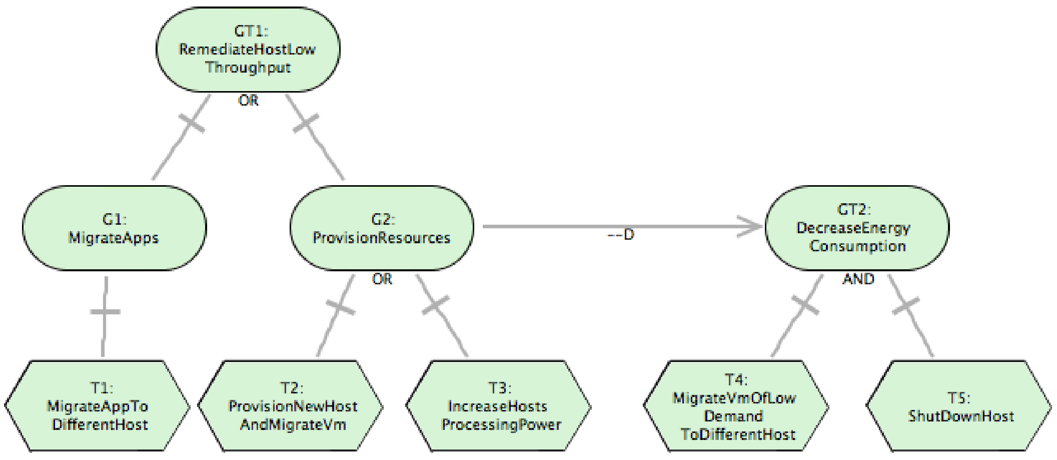

GT1: RemediateHostLow Throughput
OR
G1: MigrateApps
G2: ProvisionResources
--D
GT2: DecreaseEnergy Consumption
OR
AND
T1: MigrateAppTo DifferentHost
T2: ProvisionNewHost AndMigrateVm
T3: IncreaseHosts ProcessingPower
T4: MigrateVmOfLow Demand ToDifferentHost
T5: ShutDownHost

FIGURE 5.1: Sample Task Resolution Model

Figure 5.1 includes two goal trees: *RemediateHostLowThroughput* and *DecreaseEnergyConsumption*. We can now revisit the *Task Resolution Model* definition.

Definition 6: Given a set of *nodes* $\mathcal{N}$ and a set of *rules* $\mathcal{R}$ a *Task Resolution Model* is a tuple $TRM = \langle \mathcal{GT}, \mathcal{R}'_{\mathcal{B}} \rangle$ where $\mathcal{GT}$ is a set of *goal trees* and $\mathcal{R}'_{\mathcal{B}}$ is a set of *binary rules* whose *source* and *target* nodes belong to different goal trees.

5.2 Model Instantiation

After an Alert has been issued by the AlertService, the GoalModelSeletor component will associate this Alert with one or more goal nodes of the TRM saved in the GoalModelRepo. The selected nodes will eventually be decomposed into Tasks that will comprise the system's reconfiguration Plan. However, it is possible that the goals and tasks that the selected goals are decomposed to, are connected to each other through binary relations which also need to be taken into account when reasoning on the TRM. Additionally, it can be the case that other goals or tasks, which are not part of the selected goal trees, can contribute towards finding a resolution plan as source nodes of binary relations whose targets are goals or tasks included in the selected TRM. These nodes need to also be instantiated, otherwise the reasoning framework might ignore possible solutions to the issued Alert.

To ensure that the final TRM that the reasoning framework will reason on, by creating AND/OR rules and extracting the final cnf formula, includes all the entities that can contribute towards finding an execution Plan, we provide an algorithm which given a set of goal nodes selected by the GoalModelSelector component, returns the set of entities of the designed TRM that must be instantiated. It is then guaranteed, that all possible Tasks that could provide a solution to the monitored issue have been taken into account during the reasoning process and the compilation of the execution Plan.

The algorithm is based on a recursive traversal of the TRM, and book-keeping of the visited nodes. The structure $\mathcal{H}$ serves not only as the container of the finally instantiated TRM elements, but also as the set of nodes that the algorithm has already traversed. During the traversal of the child nodes of the set of the initially selected nodes (those provided by the GoalModelSelector) the algorithm checks for possible binary relations of type $T_b = \texttt{++S}$ whose target node is the currently visited node. If such a relation is discovered, the algorithm starts traversing the source node of the binary relation as well as its offspring nodes. When all nodes have been traversed and all source nodes have been discovered, the algorithm eventually decides which binary relations of the initial TRM are still relevant by checking if both their target and source nodes are included in the set $\mathcal{H}$. If both nodes are included, then the binary relation must also be part of the instantiated model, and thus it is added to the set $\mathcal{H}$.

Algorithm 1 Model Instantiation Algorithm

```
Require: A Task Resolution Model $TRM = \langle \mathcal{N} = \mathcal{G} \cup \mathcal{T}, \mathcal{R} = \mathcal{R_D} \cup \mathcal{R_B}, \mathcal{C} \rangle$,
    and a set of selected nodes $GN \subset \mathcal{G}$
Ensure: A set $\mathcal{H}$ containing the instantiated nodes and relations
    procedure INSTANTIATE($GN$)
        $\mathcal{H} := \emptyset$
        for all $gn \in GN$ do
            processNode($gn$)
        end for
        for all $r_b = \langle T_b,\ s,\ t \rangle \in \mathcal{R_B}$ do
            if $s \in \mathcal{H} \ \wedge \ t \in \mathcal{H}$ then
                $\mathcal{H} := \ \mathcal{H} \ \cup \ r_b$
            end if
        end for
    end procedure

    function PROCESSNODE($n$)
        if $n \in \mathcal{H}$ then return
        else
            $\mathcal{H} := \ \mathcal{H} \ \cup \ n$
            if $\exists r_b = \langle ++\mathtt{S},\ s,\ t \rangle \in \mathcal{R_B}$ then
                processNode($s$)
            end if
            if $\exists r_d = \langle T_d,\ p,\ O \rangle \in \mathcal{R_D}$ then
                for all $o \in O$ do
                    processNode($s$)
                end for
            end if
        end if
    end function
```

5.3 CNF Formula Extraction

In this section we present the transformation of a *TRM* to a CNF formula and therefore the reduction of plan resolution to SAT. The transformation process involves three steps:

1. transform the decomposition rules of each parent node into AND/OR rules
2. expand the AND/OR rules of each parent node with the side-effect rules of its source nodes
3. transform the AND/OR rules into cnf expressions

Steps 1 and 2 are both explained in table 5.1. The first rule of the table shows the rules generated by a parent node due to its decomposition relation with its offsprings. The rest of the table explains how the AND/OR rules of a parent node with a non-empty set of source nodes is formed.

Decomposition and Binary Rules	**T$_b$**	**Generated AND/OR rules**
$r_d = \langle T_d, p, O\rangle, T_d \in \{AND, OR\}$	-	$p_d \leftrightarrow T_d(o_1, o_2, \ldots), o_1, \ldots \in O$
$r_b = \langle T_b, s_i, t\rangle, s_i \in source(t)$ and $r_d = \langle T_d, t, O\rangle,\ T_d \in \{AND, OR\}$	`++S`	$t \leftrightarrow OR(p_d, p_c)$ where $p_d \leftrightarrow T_d(o_1, \ldots),\ o_1, \ldots \in O$ and $p_c \leftrightarrow OR(s_1, s_2, \ldots)$
$r_b = \langle T_b, s_i, t\rangle, s_i \in source(t)$ and $r_d = \langle T_d, t, O\rangle,\ T_d \in \{AND, OR\}$	`--D`	$t \leftrightarrow OR(p_d, p_c)$ where $p_d \leftrightarrow T_d(o_1, \ldots),\ o_1, \ldots \in O$ and $p_c \leftrightarrow OR(\neg s_1, \neg s_2, \ldots)$
$r_b = \langle T_b, s_i, t\rangle, s_i \in source(t)$ and $r_d = \langle T_d, t, O\rangle,\ T_d \in \{AND, OR\}$	`--S`	$t \leftrightarrow AND(p_d, p_c)$ where $p_d \leftrightarrow T_d(o_1, \ldots),\ o_1, \ldots \in O$ and $p_c \leftrightarrow OR(s_1, s_2, \ldots)$
$r_b = \langle T_b, s_i, t\rangle, s_i \in source(t)$ and $r_d = \langle T_d, t, O\rangle,\ T_d \in \{AND, OR\}$	`++D`	$t \leftrightarrow AND(p_d, p_c)$ where $p_d \leftrightarrow T_d(o_1, \ldots),\ o_1, \ldots \in O$ and $p_c \leftrightarrow OR(\neg s_1, \neg s_2, \ldots)$

Table 5.1: AND/OR Rule Generation

In the sample TRM of figure 5.1 the parent node GT1 generates the OR rule $GT1_d \leftrightarrow OR(A, B)$. The parent node GT2 which also serves as a target node for a binary relation, generates the rule $GT2 \leftrightarrow OR(GT2_d, GT2_c)$ where $GT2_d \leftrightarrow AND(T4, T5), GT2_c \leftrightarrow OR(\neg G2)$ corresponds to the second step of the transformation and the remaining four rows of the rule generation table.

Having created the AND/OR rules for each parent node of a TRM, step 3 involves the transformation of these rules to a cnf formula. Table 5.2 showcases the mapping between AND/OR rules and their respective cnf formula.

AND/OR Rules	**Propositional Relations**	**CNF Clauses**
$p \Leftrightarrow AND(o_1, o_2, \ldots)$	$p \rightarrow o_1, p \rightarrow o_1, \ldots, p \leftarrow (o_1 \wedge o_2 \wedge \ldots)$	$(\neg p \vee o_1) \wedge (\neg p \vee o_2) \wedge \ldots \wedge (\neg o_1 \vee \neg o_2 \vee \ldots \vee p)$
$p \Leftrightarrow OR(o_1, o_2, \ldots)$	$\neg p \rightarrow \neg o_1, \neg p \rightarrow \neg o_1, \ldots, \neg p \leftarrow (\neg o_1 \wedge \neg o_2 \wedge \ldots)$	$(\neg p \vee o_1) \wedge (\neg p \vee o_2) \wedge \ldots \wedge (o_1 \vee o_2 \vee \ldots \vee \neg p)$

Table 5.2: CNF Rule Generation

Following the last example from the sample TRM presented in 5.1, the goal tree GT2 will generate the following cnf formula:

$$(\neg GT2_c \vee GT2) \wedge (\neg GT2_d \vee GT2) \wedge (GT2_c \vee GT2_d \vee \neg GT2) \wedge$$
$$(T4 \vee \neg GT2_d) \wedge (T5 \vee \neg GT2_d) \wedge (GT2_d \vee \neg T4 \vee \neg T5) \wedge$$
$$(G2 \vee GT2_c) \wedge (\neg G2 \vee GT2_c)$$

Chapter 6

Experimental Evaluation

After describing the architecture of a framework for self-adaptive systems, its core entities and its reasoning framework, we proceed by providing experiments run on our prototype implementation of a self-adaptive system based on the proposed framework. Our self-adaptive system operates on top of the CloudSim simulation framework following a master-slave pattern, as described in Chapter 3. In the following, we explain the execution setup of both systems and present experiments that showcase how the execution plans that the self-adaptive system has devised, alternate the behavior of the CloudSim framework.

6.1 Execution Setup

6.1.1 Self-Adaptive System

For the purposes of the following set of experiments, we have configured and extended the self-adaptive framework so that it can address the issue of overbooking on virtual machines that are provisioned in the hosts of a data-center. The framework has been modified by a) defining an AlertClassifier that will generate an alert when Overbooking of a Virtual Machine is observed, b) defining an appropriate Task Resolution Model that provides a solution to the problem of Vm overbooking and assigning weights to its nodes that represent the cost-benefit of achieving them, c) implementing the Actions that correspond to the Tasks of the TRM, d) using the MaxSat module of the Sat4J framework to find an execution Plan.

AlertClassifier - Vm Overbooking

As explained in section 3.2.4, the AlertClassifier subcomponent of the self-adaptive framework is responsible for determining when a sequence of LogEvents indicates an undesirable state of the controlled system, for which the self-adaptive framework needs to reason on and provide an execution Plan. This subcomponent provides an interface with the same name which must be implemented by the developers. For our execution environment, where we wish to address the problem of Vm Overbooking, we have implemented a classifier which monitors the total overbooking of Mips per Virtual Machine (see Appendix A), and raises an Alert after the tenth time such a LogEvent is monitored.

Vm Overbooking Task Resolution Model

Figure 6.1 depicts the defined TRM and the costs assigned to its nodes. This TRM expresses that the issue of overbooking on a virtual machine can be resolved either by achieving some redistribution of load between the Virtual Machines or by provisioning new resources. As the second action introduces additional costs for managing the infrastructure (i.e. more energy consumption) it has a negative cost-benefit factor which will make it less favorable for a selection by the MaxSat solver. In the contrary, the redistribute load node has a positive cost-benefit factor so that it is preferable to the provisioning of new resources. Both of these two goals, are further decomposed to tasks, each of which has its own benefit-cost factor, that represents a relative comparison between the preference amongst them.

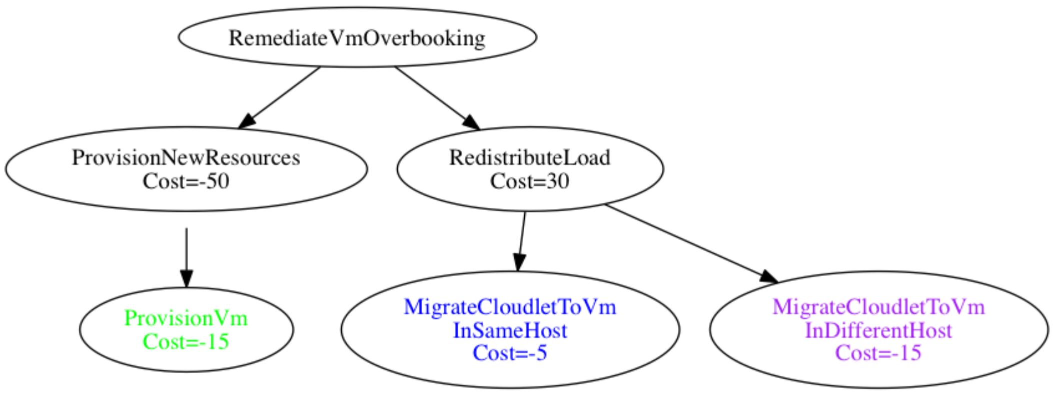

RemediateVmOverbooking
ProvisionNewResources Cost=-50
RedistributeLoad Cost=30
ProvisionVm Cost=-15
MigrateCloudletToVm InSameHost Cost=-5
MigrateCloudletToVm InDifferentHost Cost=-15

Figure 6.1: Task Resolution Model For Vm Overbooking

Actions

For each Task contained in the Vm Overbooking TRM, the corresponding Action is implemented in our framework by extending the Action abstract class and implementing the *isFeasible* and *execute* methods which we first introduced in section 4.2.11. Here we visit the functionality of these actions, to provide better understanding of the evaluation environment.

ProvisionVm: This action implements the functionality of provisioning a new Virtual Machine in the managed cloud infrastructure. The method *isFeasible* determines whether Hosts that can accommodate a new virtual machine are available. Such a Host may be in use and already serving other Virtual Machines or idle. If no Host has enough capacity for a new Virtual Machine then the isFeasible action returns false, and the corresponding Task is falsified so that the MaxSat solver cannot consider it as a possible solution. The *execute* method, contains the required functionality for the creation of a new Virtual Machine to a Host. The target Host is chosen by giving priority to Hosts who already have assigned Virtual Machines but are still able to accommodate an extra one. If no such Host exists, an idle Host is selected as the target Host.

MigrateCloudletToVmInSameHost: This action implements the functionality of migrating a Cloudlet from a source Virtual Machine to a target Virtual Machine where both are provisioned in the same Host. The *isFeasible* method returns true if a Virtual Machine whose average utilization is smaller than the average required Mips of the Cloudlet exists. The *execute* method is responsible for implementing the migration functionality which involves choosing which Cloudlet should be migrated to a different a Virtual Machine, and choosing which Virtual Machine will serve as the target Virtual Machine. It does so by implementing a best-match algorithm between the Virtual Machines' capacity and the required Mips of the Cloudlets that are executing in this Host.

MigrateCloudletToVmInDifferentHost: This action implements similar functionality with the previous one, except that in this case all the Virtual Machines of the data-center (irrespectively of their assigned Host) are considered as potential target Virtual Machines. Although these actions seem almost identical, the difference lies in performance and execution costs. Migrating a Cloudlet between different Hosts introduces latency during the transfer of data and higher utilization of the Network's bandwidth. For this reason, this action is considered to be more costful than the previous, a fact reflected in the difference between their benefit-cost factors.

Sat4J MaxSat Solver

In our implementation we have chosen to use the MaxSat module of the Sat4J solver in order to construct an execution Plan from a Hypothesis generated by an Alert. The proposed Hypothesis is transformed into a cnf formula, as described in section 5.3, and in addition to that each literal that corresponds to a node of the selected TRM, is associated with an integer value. This formula along with the weights of the literals, are provided to Sat4J. If the formula is satisfiable, a satisfying model is returned with its total cost. The truth values of the leaves of the TRM determine of which Actions the execution Plan will be comprised of.

6.1.2 CloudSim

In this section, we explain the runtime parameters of the CloudSim simulation framework which are used for our set of experiments.

Simulation Time and Step

For our evaluation process, we define each simulation session to be equal to a calendar day. The time unit of CloudSim is measured in seconds and therefore the `Simulation Limit` constant is set to $300 \times 60 \times 60\ sec = 86400\ sec = 24\ h$.

CloudSim's simulation model imposes that the execution progress of each Cloudlet is continuously updated and monitored at every simulation step. In our set of experiments the simulation step is equal to five minutes, which is a common window of monitoring cloud infrastructure. Thus, the `Simulation Step` constant is set to $300\ sec = 5\ min$.

As shown in [33], the execution time of MaxSat exceeds the `Simulation Step` for models of size bigger than 240 nodes. Therefore developers and administrators of the controlled system using our implementation of the framework which depends on MaxSat, are limited to defining a TRM consisting of less than 240 nodes.

Resource Characteristics

The cloud infrastructure resources that CloudSim treats as its core entities are Hosts and Virtual Machines. Table 6.1 showcases the characteristics of Hosts and Virtual Machines that are used for each simulation session.

	Mips	**Cores**
Host	2660	2
Vm	1000	2

Table 6.1: Host and Vm Simulation Characteristics

The distribution of a Host's Mips between the Virtual Machines that are assigned to it, is handled by the *VmScheduler* entity of the CloudSim framework. We have extended this entity so that it allows oversubscription of Virtual Machines to Hosts, meaning that any Virtual Machine can dynamically require more Mips from its Host than its initial Mips value. This allows us to observe overbooked Hosts who are unable to serve the requests of the Cloudlets that are running on their Virtual Machines.

The equivalent entity for the Virtual Machine level is the *CloudletScheduler* entity, responsible for the update of the execution of Cloudlets running on a Virtual Machine. The demand of Mips by each Cloudlets is dynamic and changes for each simulation step. The CloudletScheduler encapsulates the required functionality for the distribution of a Virtual Machine's cores and Mips to the Cloudlets. In our extended implementation, Cloudlets are served in a first come - first serve fashion, which can result in some Cloudlet entities getting their requests for Mips denied.

Cloudlet Types

As already explained, Cloudlets represent applications that are running on Virtual Machine. In our execution environment we have defined five different types of Cloudlets, depending on the demand of Mips they generate. This demand is computed at each execution step and is randomly generated as a percentage of the total Mips of the Virtual Machine that the Cloudlet is running on. In table 6.2 the different types of Cloudlets, the load they generate and the percentage of the total Cloudlets that they represent is showcased.

Type	Range of generated load (%)	% of total Cloudlets	In-Out
0	0 – 30	25	✗
1	50 – 70	10	✓
2	30 – 50	20	✗
3	30 – 50	10	✓
4	50 – 70	25	✗
5	70 – 90	10	✗

Table 6.2: Cloudlet Types

The In-Out flag determines whether a Cloudlet is active during the whole simulation session. In-Out Cloudlet types will generate load only during some part of the simulation session, and will otherwise generate 0 load. With this modeling we aim to simulate the dynamic addition or removal of Cloudlets during a simulation session.

6.2 Experiments

In this section we provide two use cases where the self-adaptive system implemented with the execution setup introduced in section 6.1.1, is used to control the CloudSim framework. In each use case we follow the setup of the CloudSim framework defined in section 6.1.2, changing the number of Hosts, Virtual Machines and Cloudlets. Finally, we use the same CloudSim setups to run the CloudSim framework without an adaptive system, and compare the results.

For each use case we provide a) its CloudSim configuration (number of Hosts, Virtual Machines, Cloudlets), b) a table presenting the types of the Cloudlets, c) a table presenting the re-deployment actions taken, d) a figure for each Virtual Machine showcasing the % of overbooking it experienced throughout an adaptive and a non - adaptive execution, and e) a figure for each Cloudlet showcasing the % of denied Mips it experienced throughout an adaptive and a non - adaptive execution. All figures contain colored vertical lines which represent the executed Actions, as shown in the Vm Overbooking Task Resolution Model of figure 6.1.

6.2.1 Use Case 1

In the first use case we choose a CloudSim configuration consisting of four Hosts, five Virtual Machines and fifteen Cloudlets. The Cloudlet types are shown in table 6.3, the re-deployment actions in table 6.4 and finally the overbooking % of Virtual Machines and the denied % of Mips per Cloudlet are shown in figures 6.2 and 6.3 respectively.

In the adaptive execution, a reduction on Vm Overbooking is observed, compared to the non - adaptive execution, in all Virtual Machines except for virtual machine 2, which experiences increased overbooking in the first part of the simulation. This behavior is mainly - but not only - observed due to the addition of Virtual Machines (actions 5, 12, 15) to hosts who were already running, yet they were not fully utilised. However we can see that the self-adaptive system has chosen to postpone these actions until the 84000.1 sec of the simulation, trying to find a solution with fewer costs. This is reflected in actions 0, 1, 2, 3, 4, all of which aim to redistribute the load on the current infrastructure without the provisioning additional resources. These actions manage to improve the overbooking situation on all Virtual Machines except for Virtual Machines 2 and 3, which experiences slightly increased overbooking after the redistribution actions have been executed. The improvement of performance of Virtual Machine 2, can no longer be achieved with load redistribution , as the best matches have already been found and used, and therefore the adaptive system decides to provision new resources by executing action 5.

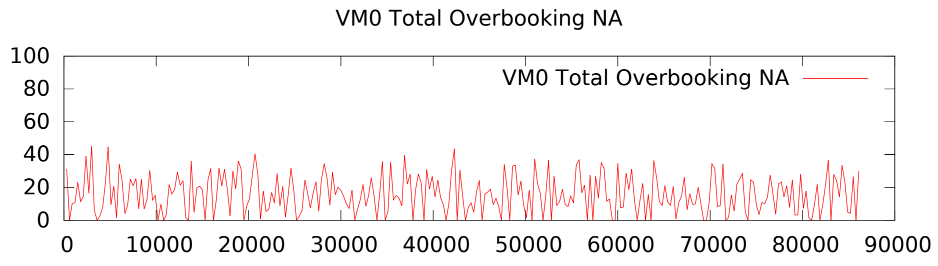
After the first provisioning of a new Virtual Machine another set of re-mediation actions are executed until the In-Out Cloudlets kick in after approximately 35000 sec. This new demand created by the In-Out Cloudlets forces re-mediation actions 14, 15, 16, which showcase the adaptivity of the system, which is able to not only redistribute its initial load, but also accommodate future situations.

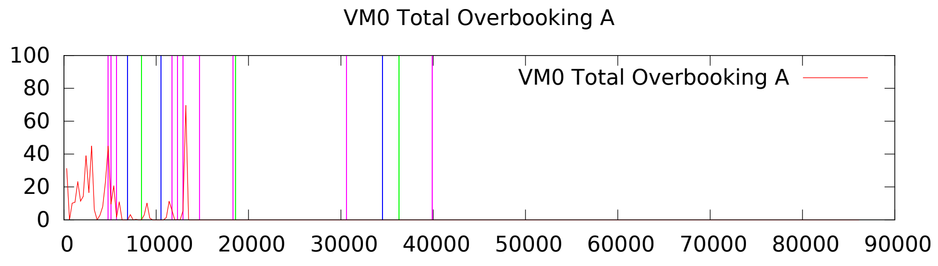
Cloudlet Type	**Cloudlet Ids**
0	0, 1, 2
1	3
2	4, 5, 6
3	7
4	8, 9, 10
5	11

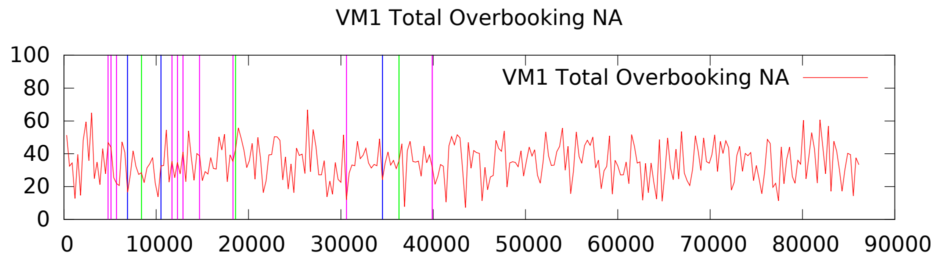
Table 6.3: Use Case 1 Cloudlets

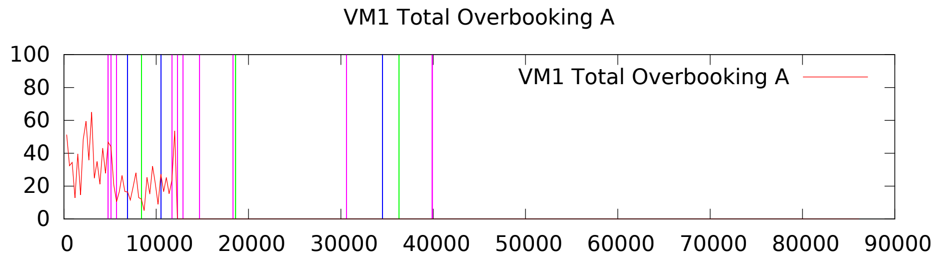
Action id	**Time**	**Action**
0	4800.1	Migrating Cloudlet 4 from Vm 4 to VM 2
1	5100.1	Migrating Cloudlet 9 from Vm 4 to VM 2
2	5100.1	Migrating Cloudlet 1 from Vm 1 to VM 2
3	5700.1	Migrating Cloudlet 0 from Vm 0 to VM 2
4	6900.1	Migrating Cloudlet 2 from Vm 2 to VM 3
5	8400.1	Creating Vm on Datacenter 3
6	10500.1	Migrating Cloudlet 1 from Vm 2 to VM 3
7	11700.1	Migrating Cloudlet 6 from Vm 1 to VM 4
8	12300.1	Migrating Cloudlet 3 from Vm 3 to VM 5
9	12900.1	Migrating Cloudlet 5 from Vm 0 to VM 4
10	14700.1	Migrating Cloudlet 7 from Vm 2 to VM 5
11	18300.1	Migrating Cloudlet 4 from Vm 2 to VM 5
12	18900.1	Creating Vm on Datacenter 3
13	30600.1	Migrating Cloudlet 8 from Vm 3 to VM 6
14	34500.1	Migrating Cloudlet 3 from Vm 5 to VM 6
15	36600.1	Creating Vm on Datacenter 3
16	39900.1	Migrating Cloudlet 8 from Vm 6 to VM 7

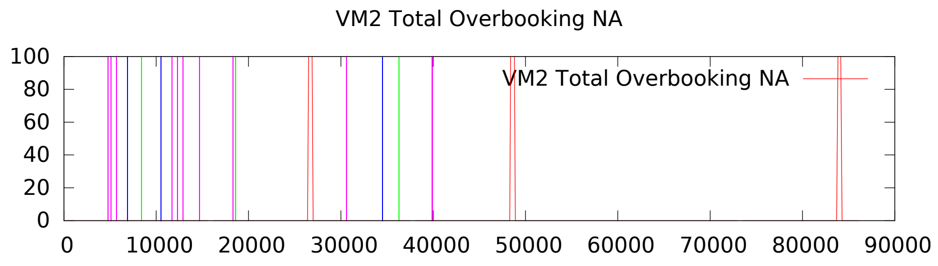
Table 6.4: Use Case 1 Actions

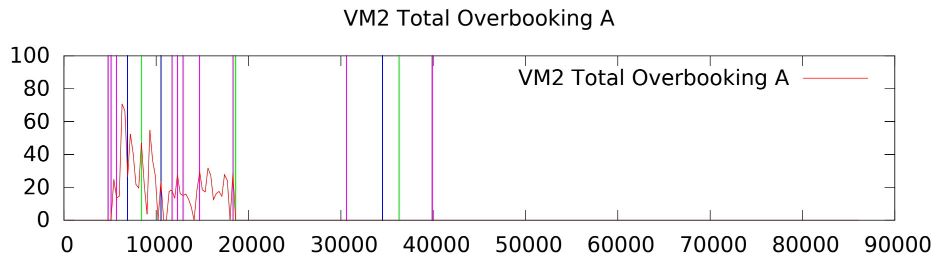

VM0 Total Overbooking NA
VM0 Total Overbooking NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM0 Total Overbooking A
VM0 Total Overbooking A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM1 Total Overbooking NA
VM1 Total Overbooking NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM1 Total Overbooking A
VM1 Total Overbooking A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM2 Total Overbooking NA
VM2 Total Overbooking NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM2 Total Overbooking A
VM2 Total Overbooking A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

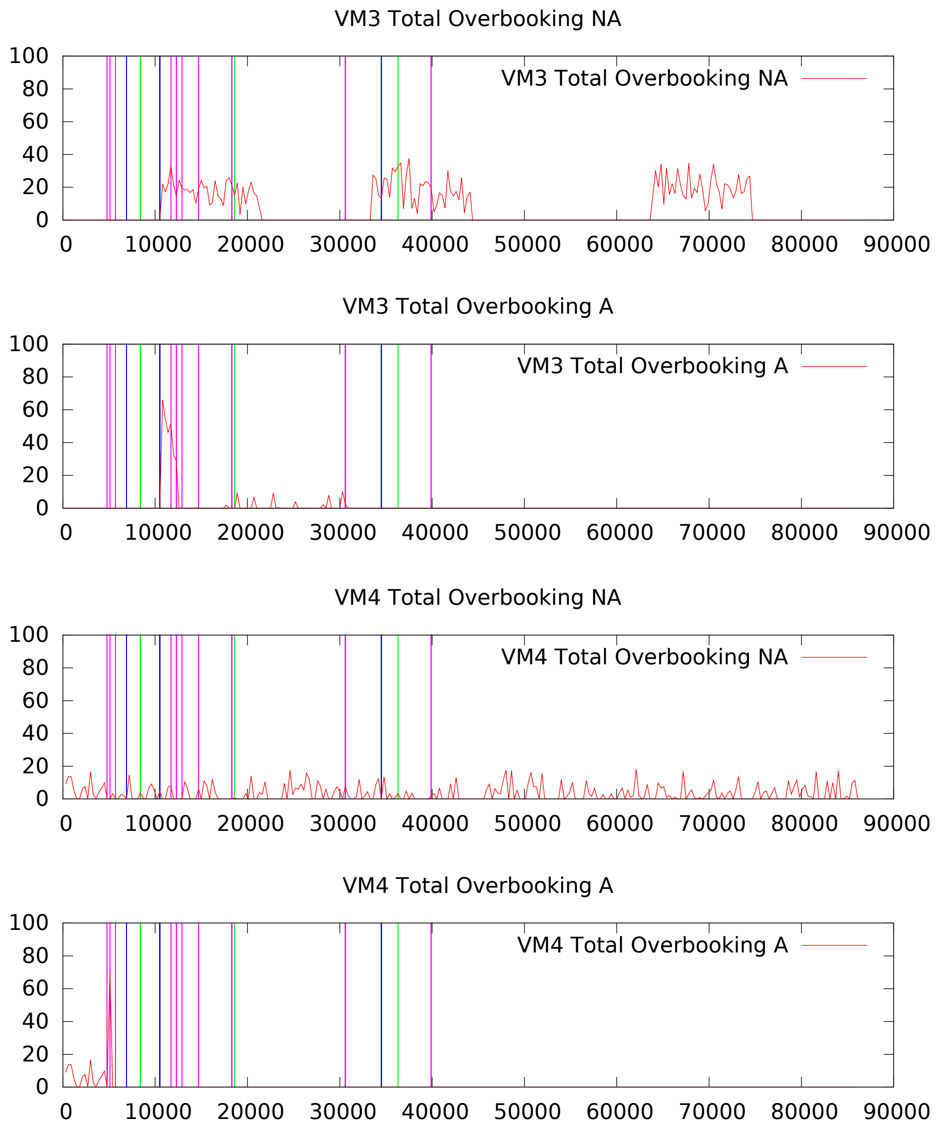
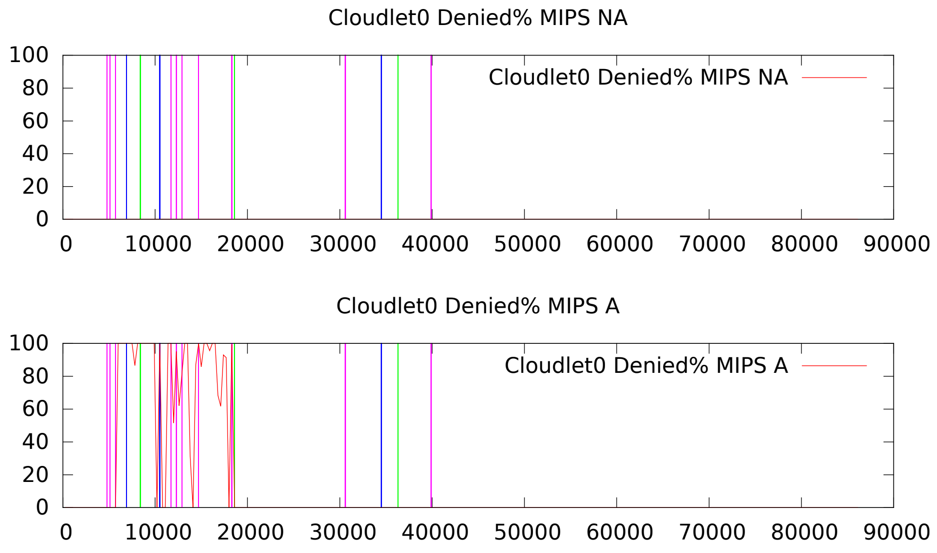
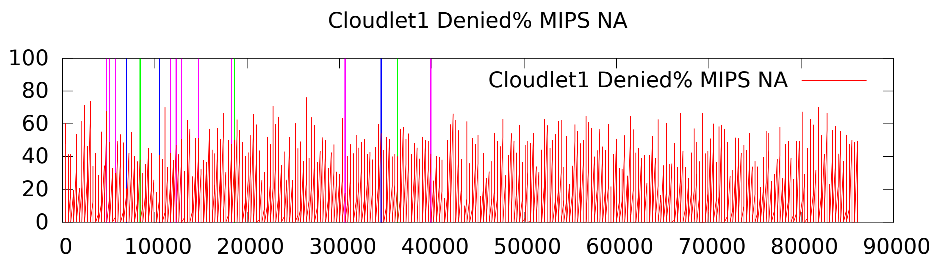
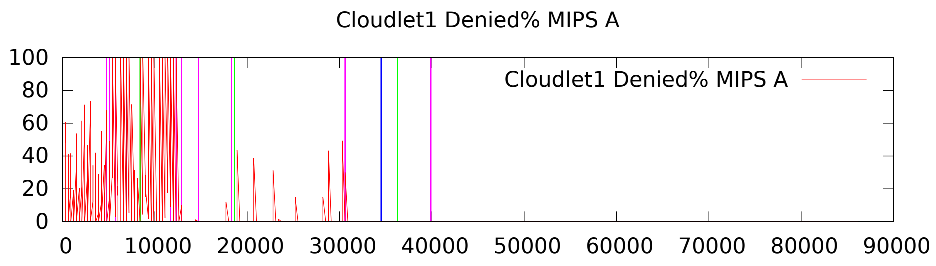
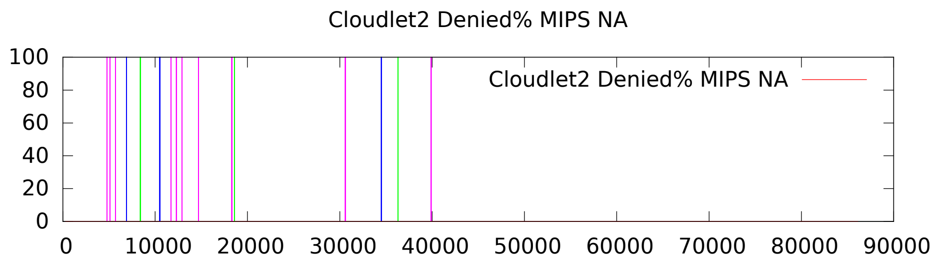


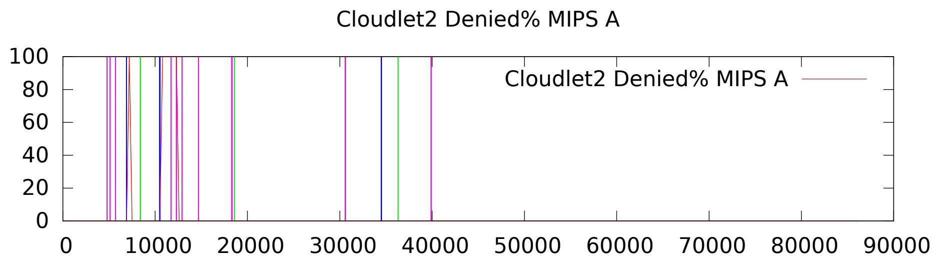
FIGURE 6.2: Use Case 1 Vm Overbooking

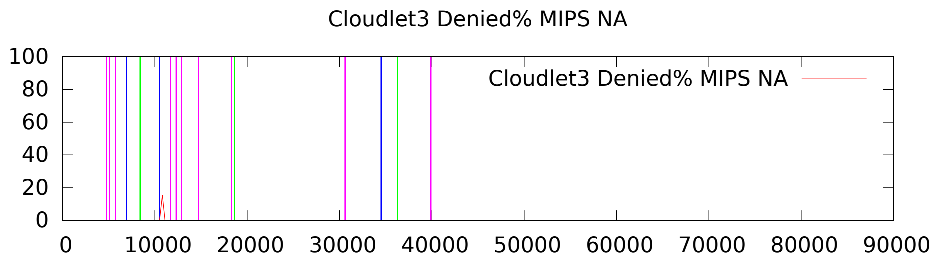


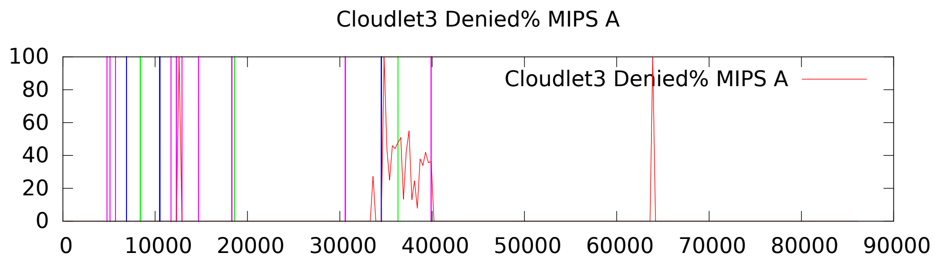

Cloudlet1 Denied% MIPS NA
Cloudlet1 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

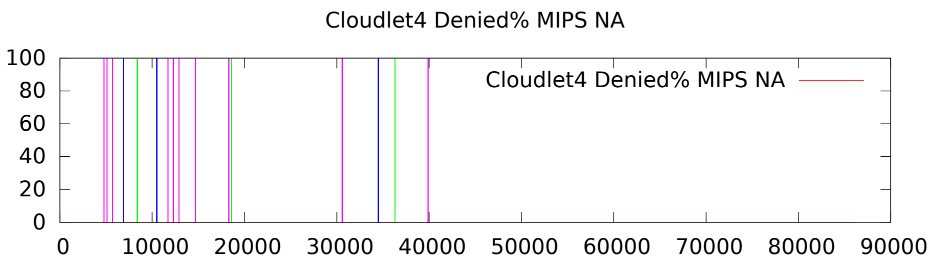

Cloudlet1 Denied% MIPS A
Cloudlet1 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

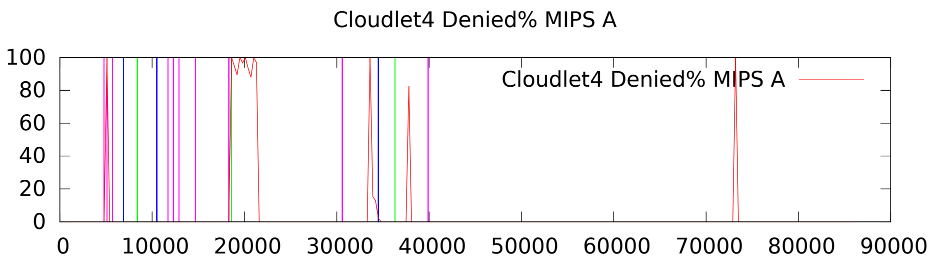

Cloudlet2 Denied% MIPS NA
Cloudlet2 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

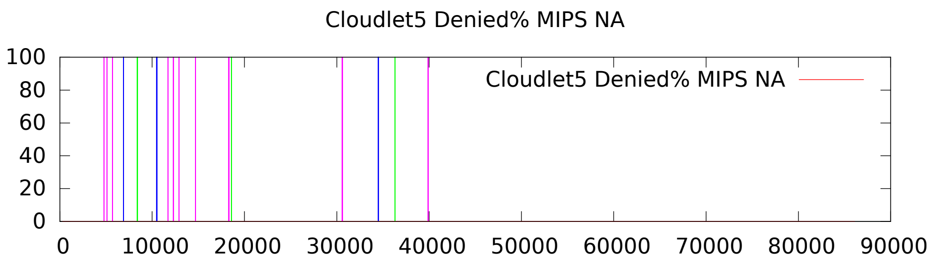

Cloudlet2 Denied% MIPS A
Cloudlet2 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

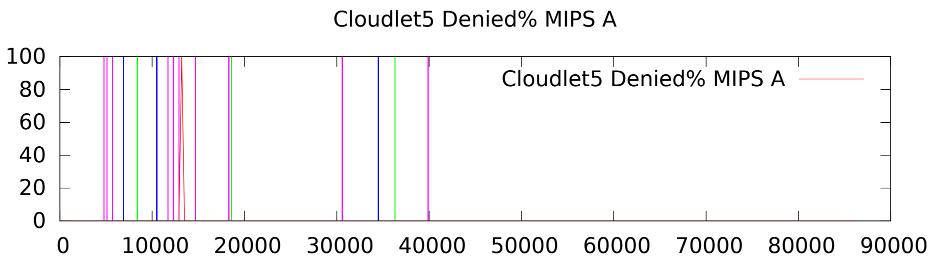

Cloudlet3 Denied% MIPS NA
Cloudlet3 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

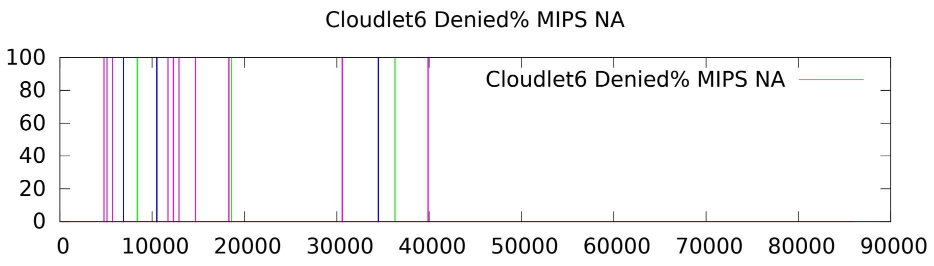

Cloudlet3 Denied% MIPS A
Cloudlet3 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

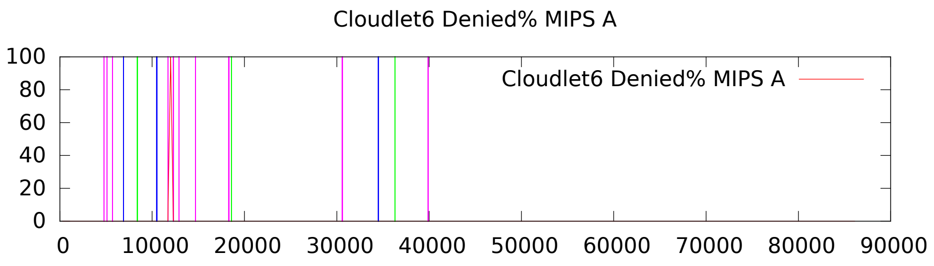

Cloudlet4 Denied% MIPS NA
Cloudlet4 Denied% MIPS NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

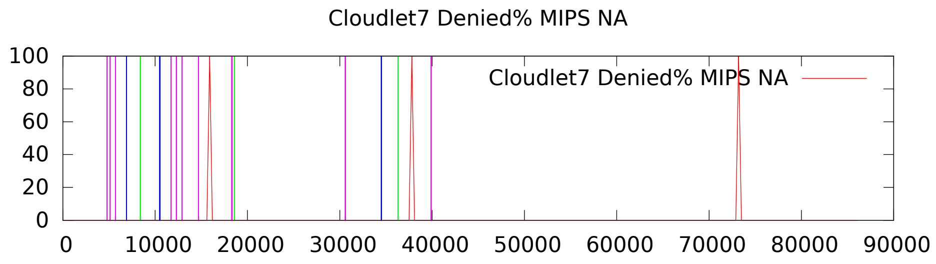

Cloudlet4 Denied% MIPS A
Cloudlet4 Denied% MIPS A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

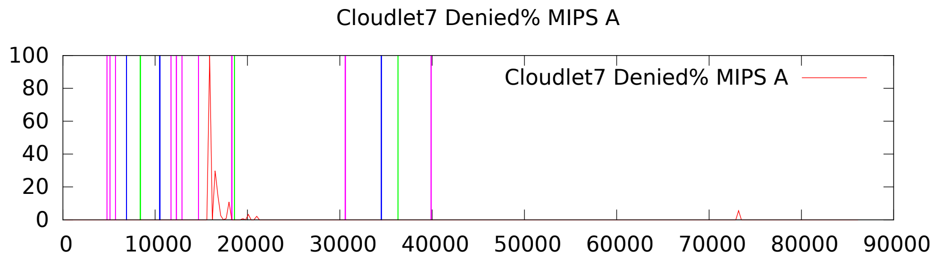

Cloudlet5 Denied% MIPS NA
Cloudlet5 Denied% MIPS NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

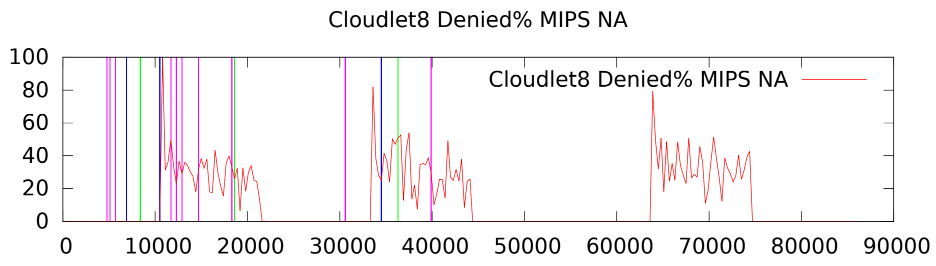

Cloudlet5 Denied% MIPS A
Cloudlet5 Denied% MIPS A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

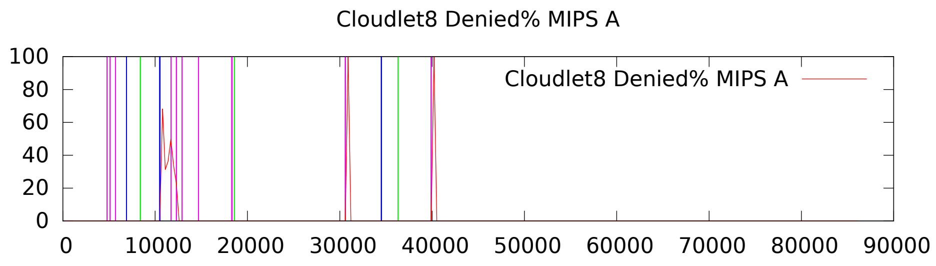

Cloudlet6 Denied% MIPS NA
Cloudlet6 Denied% MIPS NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

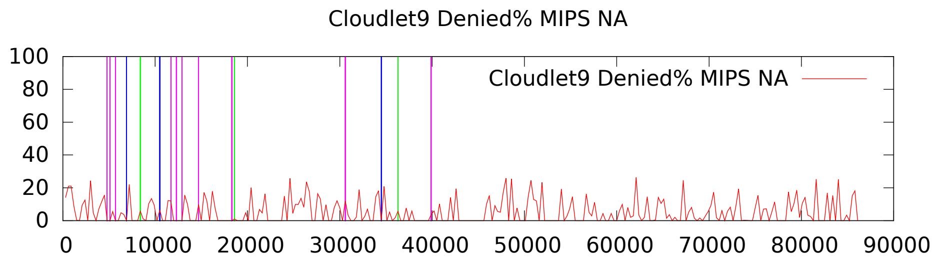

Cloudlet6 Denied% MIPS A
Cloudlet6 Denied% MIPS A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

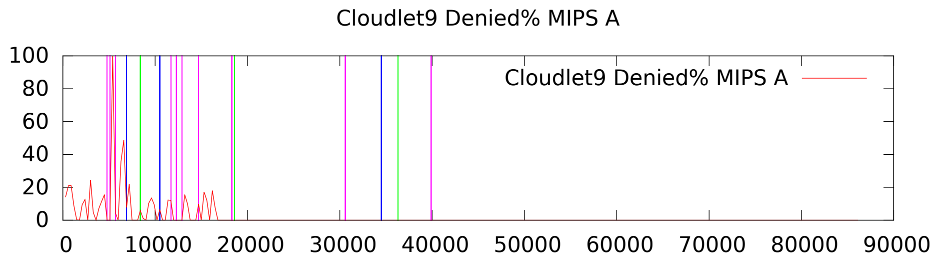

Cloudlet7 Denied% MIPS NA
Cloudlet7 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet7 Denied% MIPS A
Cloudlet7 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet8 Denied% MIPS NA
Cloudlet8 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet8 Denied% MIPS A
Cloudlet8 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet9 Denied% MIPS NA
Cloudlet9 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet9 Denied% MIPS A
Cloudlet9 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

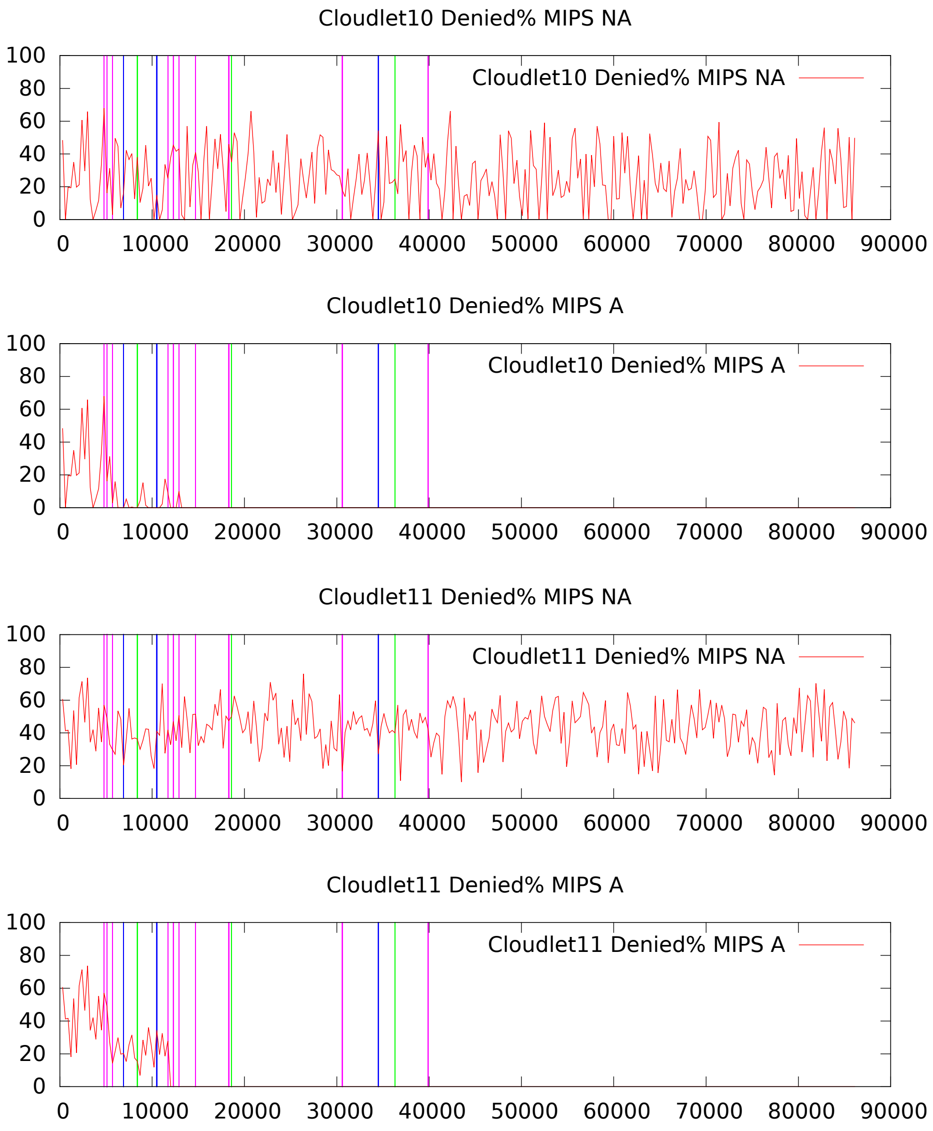


FIGURE 6.3: Use Case 1 Cloudlet Denied Mips

6.2.2 Use Case 2

In the second use case, a similar CloudSim configuration consisting of four Hosts, six Virtual Machines and seventeen Cloudlets is chosen and simulated.

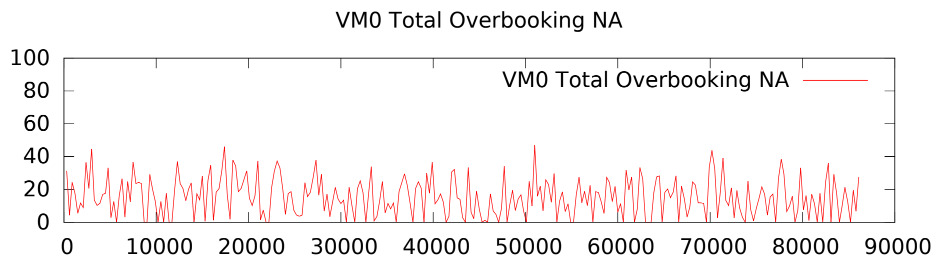
This use case is different to the first one with regards to the available resources that the self-adaptive system can provision. After a virtual machine is created, the system is limited only to load redistribution actions. As a result, some virtual machine keep experiencing overbooking during the adaptive execution, although in much lower percentages.

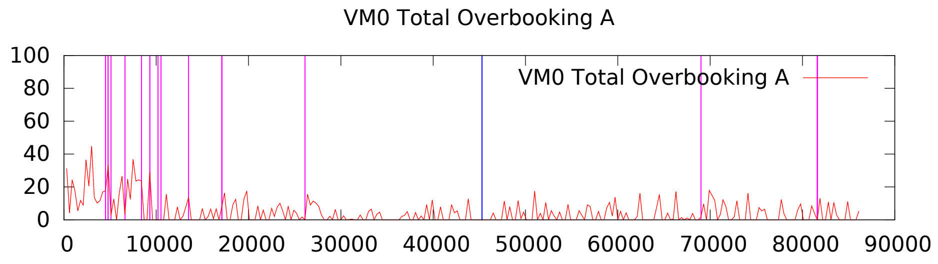
Cloudlet Type	**Cloudlet Ids**
0	0, 1, 2, 3
1	4
2	5, 6, 7
3	8
4	9, 10, 11, 12
5	13

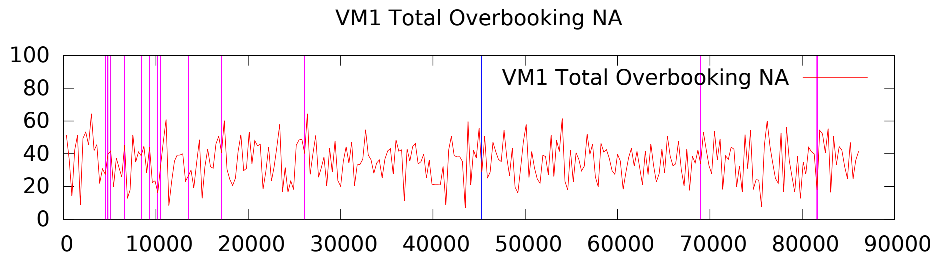
Table 6.5: Use Case 2 Cloudlets

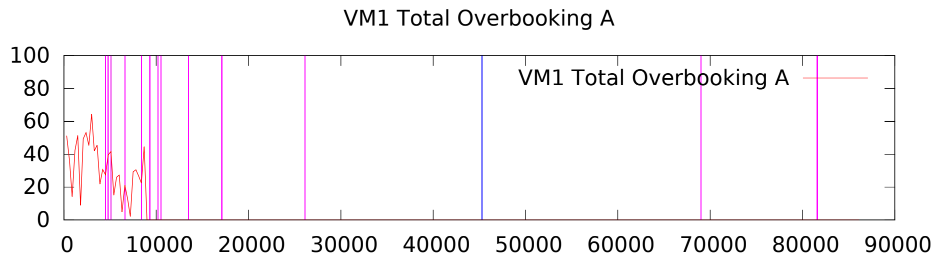
Action id	**Time**	**Action**
1	4500.1	Migrating Cloudlet 5 from Vm 5 to VM 2
2	4800.1	Migrating Cloudlet 11 from Vm 5 to VM 2
3	5100.1	Migrating Cloudlet 1 from Vm 1 to VM 2
4	5100.1	Creating Vm on Datacenter 3
5	6600.1	Migrating Cloudlet 2 from Vm 2 to VM 6
6	8400.1	Migrating Cloudlet 7 from Vm 1 to VM 6
7	8400.1	Migrating Cloudlet 4 from Vm 4 to VM 6
8	9300.1	Migrating Cloudlet 0 from Vm 0 to VM 2
9	10200.1	Migrating Cloudlet 1 from Vm 2 to VM 5
10	10500.1	Migrating Cloudlet 2 from Vm 6 to VM 2
11	13500.1	Migrating Cloudlet 8 from Vm 2 to VM 5
12	17100.1	Migrating Cloudlet 5 from Vm 2 to VM 5
13	26100.1	Migrating Cloudlet 0 from Vm 2 to VM 5
14	45300.1	Migrating Cloudlet 1 from Vm 5 to VM 4
15	69000.1	Migrating Cloudlet 0 from Vm 5 to VM 6
16	81600.1	Migrating Cloudlet 0 from Vm 6 to VM 5

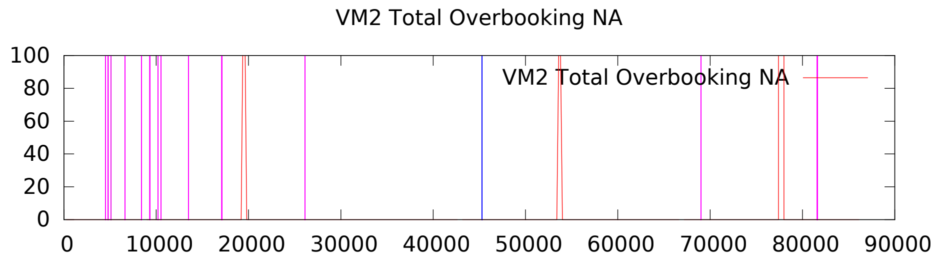
Table 6.6: Use Case 1 Actions

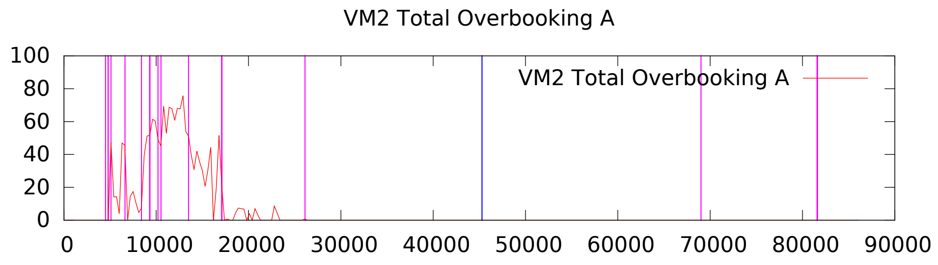

VM0 Total Overbooking NA
VM0 Total Overbooking NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM0 Total Overbooking A
VM0 Total Overbooking A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM1 Total Overbooking NA
VM1 Total Overbooking NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM1 Total Overbooking A
VM1 Total Overbooking A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM2 Total Overbooking NA
VM2 Total Overbooking NA
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


VM2 Total Overbooking A
VM2 Total Overbooking A
100
80
60
40
20
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

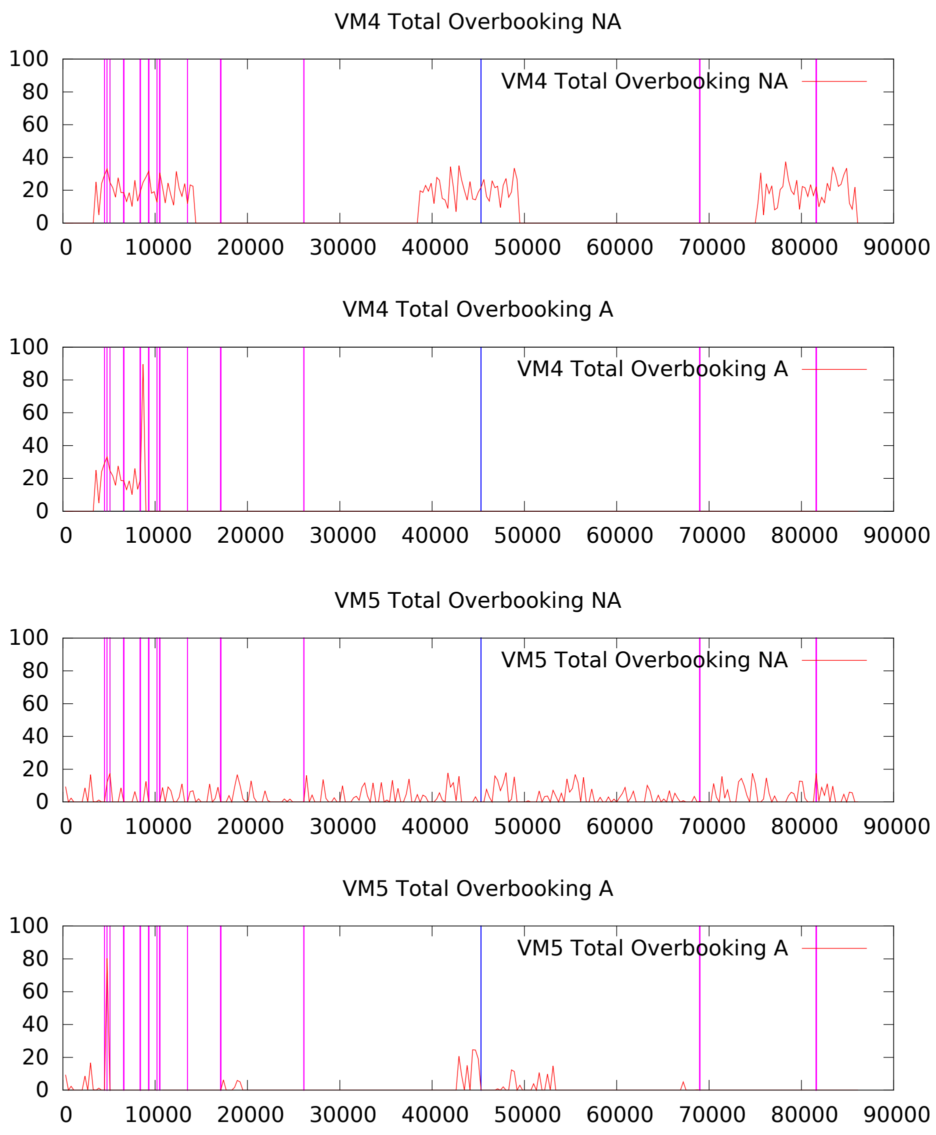
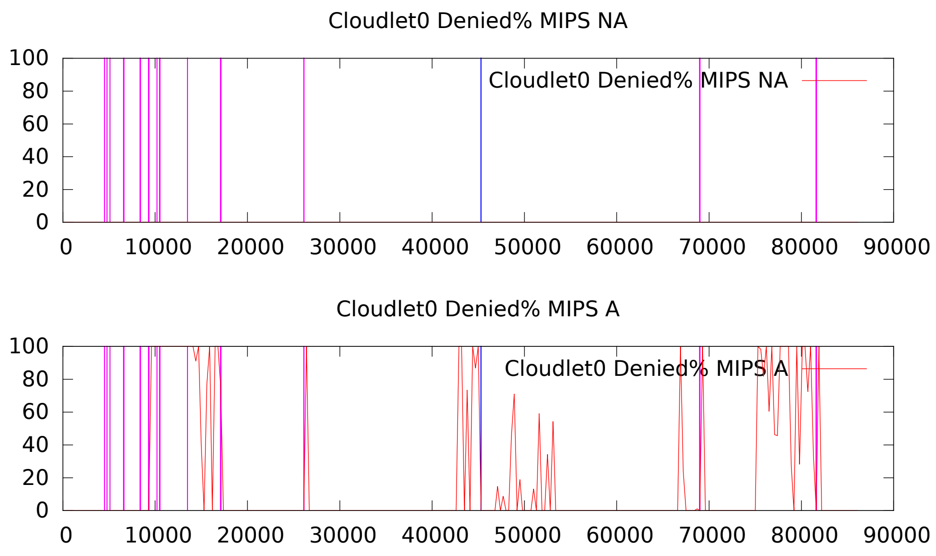
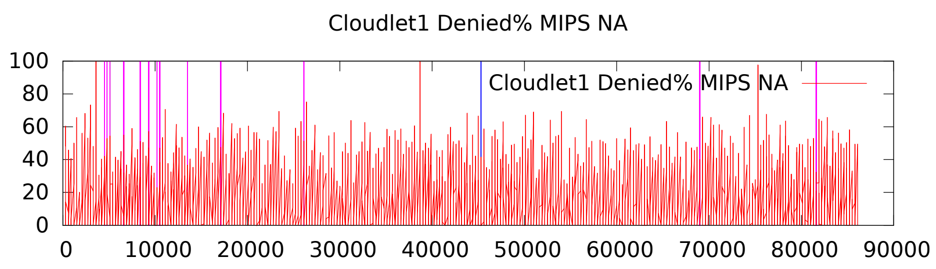
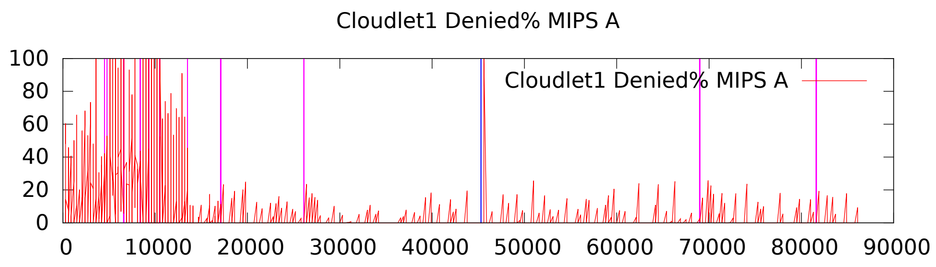
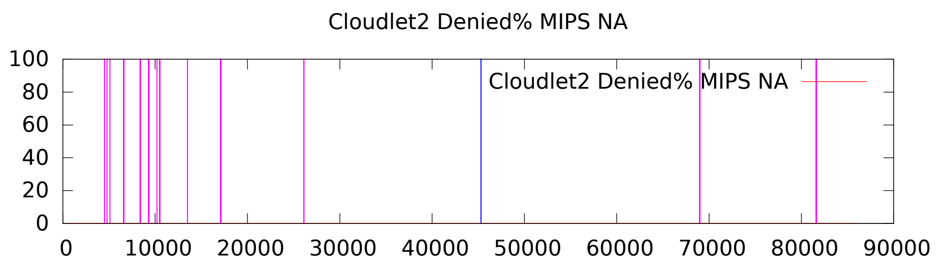


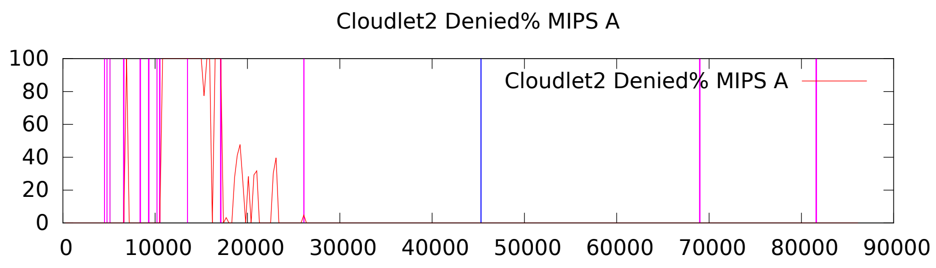
Figure 6.4: Use Case 2 Vm Overbooking

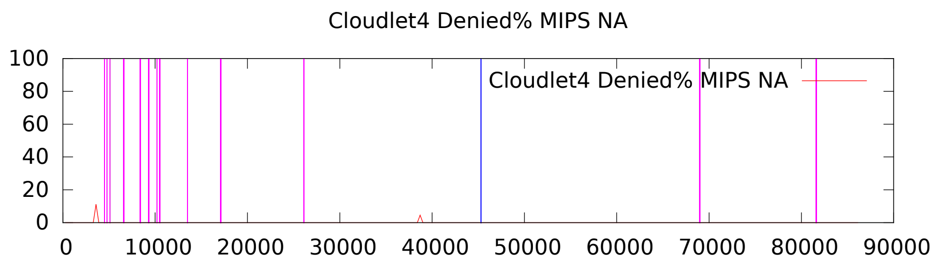


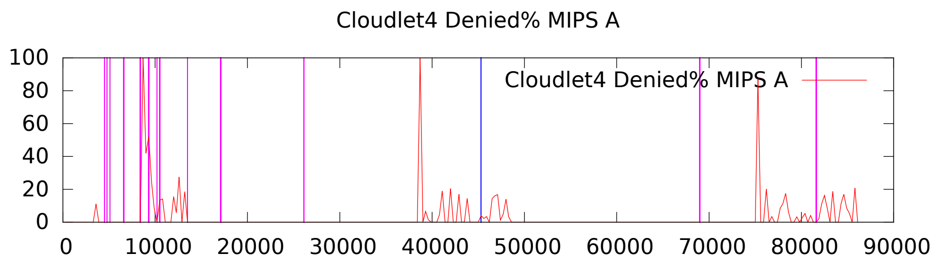

Cloudlet1 Denied% MIPS NA
Cloudlet1 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet1 Denied% MIPS A
Cloudlet1 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

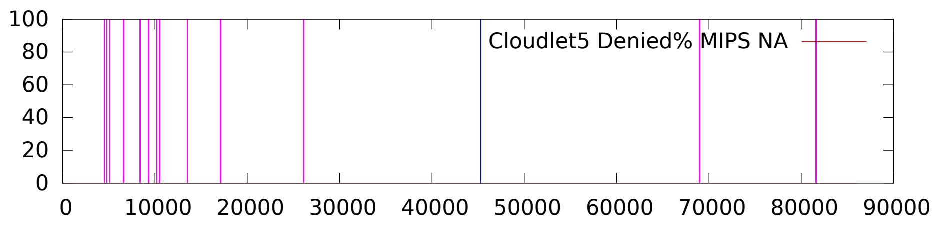

Cloudlet2 Denied% MIPS NA
Cloudlet2 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet2 Denied% MIPS A
Cloudlet2 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

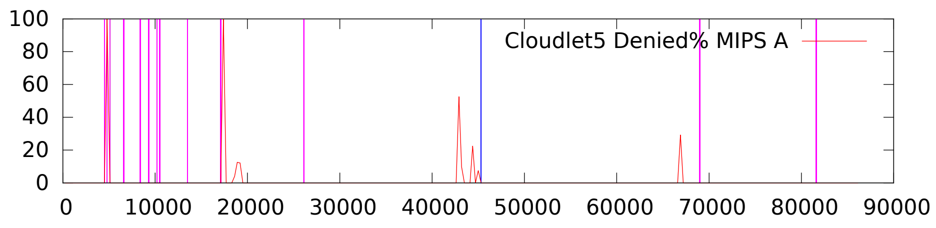

Cloudlet4 Denied% MIPS NA
Cloudlet4 Denied% MIPS NA
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


Cloudlet4 Denied% MIPS A
Cloudlet4 Denied% MIPS A
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

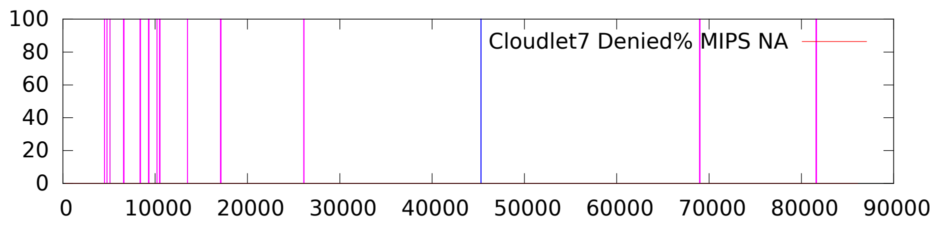
Cloudlet5 Denied% MIPS NA



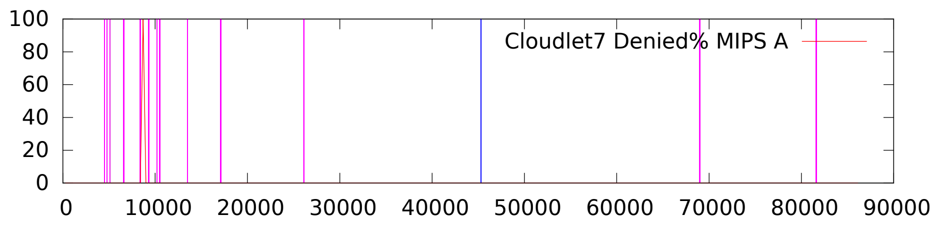
Cloudlet5 Denied% MIPS A



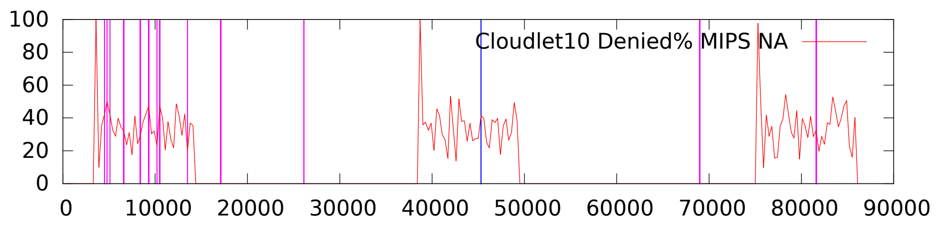
Cloudlet7 Denied% MIPS NA



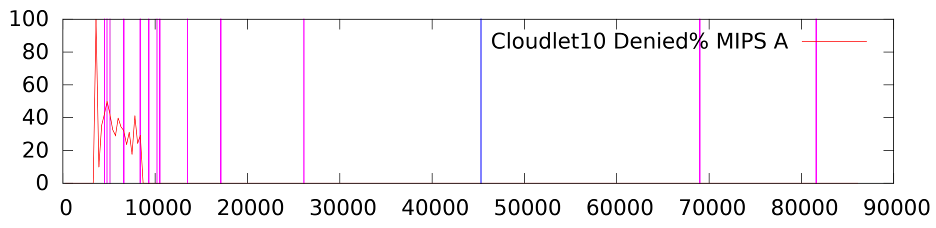
Cloudlet7 Denied% MIPS A



Cloudlet10 Denied% MIPS NA



Cloudlet10 Denied% MIPS A



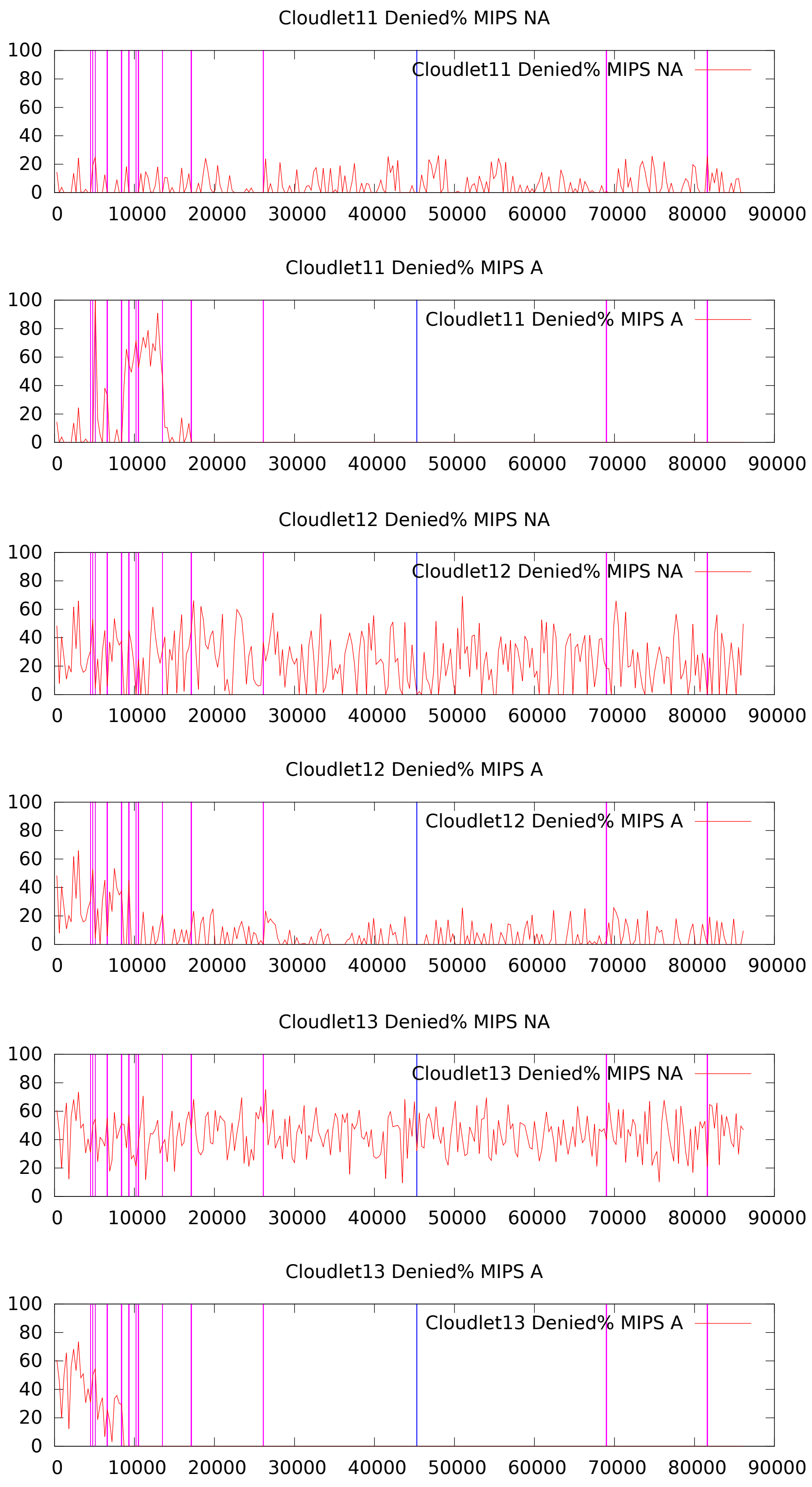


FIGURE 6.5: Use Case 2 Cloudlet Denied Mips

Chapter 7

Conclusion

7.1 General Remarks

In this thesis, we have presented a framework for self-adaptive system re-deployment. Our work aims to identify the core components of a self-adaptive and autonomous system by defining an architecture that implements the autonomic control loop and uses a reasoning mechanism based on high level models. Developers who extend our proposed framework, can intervene and extend all the parts of the MAPE-K loop through extensible interfaces and rely on a reasoning framework that will compile a re-mediation plan for the controlled system based on their preferences.

Our reasoning framework utilizes high level models that associate functional properties of the controlled system with tasks that are executed on the latter. Its reasoning process is based on the transformation of these models to a planning problem and essential reduction of the latter to a satisfiability problem. This is achieved through the rule generation we propose and the extraction of the corresponding cnf formula. The satisfied tasks of the cnf formula constitute the re-mediation plan of the self-adaptive system.

With management of cloud infrastructure being an interesting application domain for self-adaptive systems, we have developed a concrete implementation of our proposed framework and used it as a self-adaptive system on top of the CloudSim simulation framework. We have defined concrete Task Resolution Models and their corresponding Actions for the former, and extended the reporting mechanisms of the latter with additional performance metrics.

Among existing approaches for the design of adaptive systems, the work presented in this thesis is more closely related to the following two:

- **Requirements based** approaches, where Requirements Engineering techniques are extended in order to represent the requirements of adaptation and the uncertainty of the environment in which the system operates. *Zanshin* is an Requirements Engineering-based framework for the design of adaptive systems that exploits concepts of Control Theory to design adaptive software systems[34] and makes the elements of the feedback loop that provide adaptivity first class citizens in its requirements models[35].

- **Architecture based** approaches, where an architectural model that shows system components and how they communicate amongst themselves through connectors is proposed. Such proposals can include the runtime software infrastructure on top of which to build the adaptive system, taking care of its adaptation rules and how to evolve its models. The Rainbow framework[36] is a prominent architecture- based approach for the design of adaptive systems. According to the architectural model proposed, rules are used to monitor the operational conditions of the system and define actions to be taken if the conditions are unfavorable, using utility theory.

Our framework lies in between these two approaches. It bares similarities with the Requirements based approach through the definition of Task Resolution Models which define the requirements of the system that must be satisfied during its execution. Contrary to Zanshin, our system does not require strategies to be specified in an additional manner, as our Task Resolution Models define which actions could/should be taken for the requirements to be met. When compared to the Architecture base approach, our framework also enforces the definition of components that communicate with each other and can potentially involve the runtime infrastructure of the controlled system. However, the re-mediation plan we compile is not based on utility theory, but on the Task Resolution Model devised by the developers.

7.2 Future Work

Our proposed framework as well as the extensions that we have implemented in the CloudSim framework provide a fertile ground for future work. These tasks can be grouped in two directions, the first one related with the reasoning framework of the self-adpative framework, and the second related with our concrete prototype implementation of the self-adaptive system.

Our reasoning framework consisting of Task Resolution Models, an extention of Goal Models, only considers the adaptation process from the perspective of the self-adaptive and the controlled system. Differently put, it is unaware of other systems that the latter might interact with and their stakeholder's goals. One solution to this issue would be to extend our current models with agents and commitments, as they are described in[37]. This would allow for modeling of interesting interactions between different stakeholders such as vendors providing compensation to clients whose service agreement has been violated and being able to reason wether this behavior is more beneficiary for them.

Moving to implementation related future tasks, our current implementation of the self-adaptive system relies on a MaxSat based approach for the compilation of execution plans. While MaxSat is a complete algorithm and guarantees that the solution provided is optimal, it gets impractical for models of big size as shown in[33]. Therefore, a possible direction for future work would be incorporating alternative (possibly incomplete) algorithms for the compilation of the execution plans, such as the one presented in[33].

Finally, there is much room for improvement and optimization in the actions that the self-adaptive system executes. For example, algorithms for Vm consolidation could be used. In the same context, better diagnosis of issues of the controlled system could be achieved by incorporating trend-analysis methods. These methods would provide teh self-adaptive system with more meaningful Alerts and avoid re-mediation Plans that occur only due to some temporal abnormalities of the system.

Appendix A

CloudSim Metrics and KPIs

Table A.1: Cloudlet Metrics and KPIs

Metrics	KPIs
Cores	
required cores	
CPU	
total requested mips	total cpu throughput ratio (total allocated mips/total requested mips*%)
total allocated mips	per core cpu throughput (per-core allocated mips/per-core requested mips*%)
requested mips per core	active avg. of total\per-core cpu throughput per session
allocated mips per core	range of total\per core cpu throughput per session
max\min total requested mips	total denied mips pct. ((requested mips - allocated mips)/requested mips *%)
max\min requested mips per core	per-core denied mips pct.((required mips per-core - allocated mips)/required mips *%)
	active avg. of total\per-core denied mips pct. per session
	range of total\per core denied mips pct. per session
	range of total\per core requested mips per session
	delta of cpu throughput between cores
	delta of requested mips between cores

Continued on next page

Table A.1 – continued from previous page

Metrics	**KPIs**
IO	
requested iops	iops throughput (allocated iops/requested iops*%)
allocated iops	active avg. of iops throughput
max\min requested iops	range of iops throughput per session
	denied iops pct. ((requested iops - allocated iops)/iops requested*%)
	active avg. of denied iops pct.
	range of denied iops pct. per session
	range of iops requested per session
Memory	
requested ram	memory throuhgput (ram allocated/ram requested*%)
allocated ram	active avg. of memory throughput
max\min requested ram	range of memory throughput per session
	denied memory pct. ((requested ram - allocated ram)/ram requested*%)
	active avg. of denied memory pct.
	range of denied memory pct. per session
	range of ram requested per session
Network	
requested bw	network throuhgput (bw allocated/bw requested*%)
allocated bw	active avg. of network throughput
max\min requested bw	range of network throughput per session
	denied network pct. ((requested ram - allocated ram)/ram requested*%)
	active avg. of denied network pct.
	range of denied network pct. per session
	range of bw requested per session
Migration	
	cost of migration to VM in the same Host
	cost of migration to Vm in different Host
	benefit of migration to Vm in the same Host
	benefit of migration to Vm in different Host

Table A.2: VM Metrics and KPIs

Metrics	**KPIs**
Cores	
total cores	pct. of idle cores
used cores	pct. of used cores
idle cores	active avg. of idle cores
required cores per cloudlet	active avg. cores requested by cloudlets
CPU	
total mips	pct. of total\per-core utilization of mips
total allocated mips to cloudlets	pct. of total\per-core underutilization of mips
total requested mips by cloudlets	pct. of total\per-core overbooking of mips
mips per core	active avg. pct. of total\per-core utilization of mips
total allocated mips per core to cloudlets	delta of pct. utilization of mips between cores
total requested mips per core by cloudlets	range of pct. of total\per-core utilization per session
total requested mips per cloudlet	pct. of total\per-core utlization of mips per cloudlet
total allocated mips per cloudlet	delta of pct. of total\per-core utilization between cloudlets
requested mips per core per cloudlet	range of pct. of total\per-core utilization per cloudlet through session
allocated mips per core per cloudlet	
number of cloudlets running on vm	
IO	
total iops	pct. of iops utilization
total allocated iops	pct. of iops underutilization
total requested iops	pct. of iops overbooking
iops requested per cloudlet	active avg. of pct. of iops utilization

Continued on next page

Table A.2 – continued from previous page

Metrics	**KPIs**
iops allocated per cloudlet	range of pct. of iops utilization
max\min total iops requested	pct. of iops utilization per cloudlet
	delta of pct. of iops utilization between cloudlets
	range of pct. of iops utilization per cloudlet
Memory	
total ram	pct. of ram utilization
total allocated ram	pct. of ram underutilization
total requested ram	pct. of ram overbooking
ram requested per cloudlet	active avg. of pct. of ram utilization
ram allocated per cloudlet	range of pct. of ram utilization
max\min total ram requested	pct. of ram utilization per cloudlet
	delta of pct. of ram utilization between cloudlets
	range of pct. of ram utilization per cloudlet
Network	
total bw	pct. of bw utilization
total allocated bw	pct. of bw underutilization
total requested bw	pct. of bw overbooking
bw requested per cloudlet	active avg. of pct. of bw utilization
bw allocated per cloudlet	range of pct. of bw utilization
max\min total bw requested	pct. of bw utilization per cloudlet
	delta of pct. of bw utilization between cloudlets
	range of pct. of bw utilization per cloudlet
Migration	

Continued on next page

Table A.2 – continued from previous page

Metrics	KPIs
	cost of migration to Host in the same Datacenter cost of migration to Vm in different Datacenter benefit of migration to Vm in the same Datacenter benefit of migration to Vm in different Datacenter

Table A.3: Host Metrics and KPIs

Metrics	KPIs
Cores	
total cores	pct. of idle cores
used cores	pct. of used cores
idle cores	active avg. of idle cores
required cores per VM	active avg. cores requested by cloudlets
CPU	
total mips	pct. of total\per-core utilization of mips
total allocated mips to VMs	pct. of total\per-core underutilization of mips
total requested mips by VMs	pct. of total\per-core overbooking of mips
mips per core	active avg. pct. of total\per-core utilization of mips
allocated mips per core to VMs	delta of pct. utilization of mips between cores
requested mips per core by VMs	range of pct. of total\per-core utilization per session
total requested mips per VM	pct. of total\per-core utlization of mips per VM
total allocated mips per VM	delta of pct. of total\per-core utilization between vms
requested mips per core per VM	range of pct. of total\per-core utilization per VM through session

Continued on next page

Table A.3 – continued from previous page

Metrics	**KPIs**
allocated mips per core per VM	
number of VMs running on host	
Memory	
total ram	pct. of ram utilization
total allocated ram	pct. of ram underutilization
total requested ram	pct. of ram overbooking
ram requested per VM	active avg. of pct. of ram utilization
ram allocated per VM	range of pct. of ram utilization
	pct. of ram utilization per cloudlet
max\min total ram requested	delta of pct. of ram utilization between VMs
ram requested per cloudlet	range of pct. of ram utilization per VM
ram allocated per cloudlet	
Network	
total bw	pct. of bw utilization
total allocated bw	pct. of bw underutilization
total requested bw	pct. of bw overbooking
bw requested per vm	active avg. of pct. of iops utilization
bw allocated per cloudlet	range of pct. of iops utilization
max\min total bw requested	pct. of bw utilization per cloudlet
	delta of pct. of bw utilization between VMs
	range of pct. of bw utilization per VM

Bibliography

[1] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[2] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[3] International Business Machines Corporation. *An architectural blueprint for Autonomic Computing.* IBM whitepaper. IBM Corporation, International Technical Support Organization, 2005.

[4] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting dependable systems*, pages 61–89. Springer, 2003.

[5] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer, 2009.

[6] Rogrio De Lemos, Holger Giese, Hausi A Mller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[7] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[8] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John Wiley & Sons, 2007.

[9] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[10] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. *SIGSOFT Softw. Eng. Notes*, 30 (4):1–7, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083075. URL http://doi.acm.org/10.1145/1082983.1083075.

[11] IBM Redbooks and International Business Machines Corporation. International Technical Support Organization. *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM redbooks. IBM Corporation, International Technical Support Organization, 2004. ISBN 9780738498058. URL http://books.google.fr/books?id=XHeoSgAACAAJ.

[12] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[13] Sotirios Liaskos, Shakil M. Khan, Marin Litoiu, Marina Daoud Jungblut, Vyacheslav Rogozhkin, and John Mylopoulos. Behavioral adaptation of information systems through goal models. *Inf. Syst.*, 37(8):767–783, 2012.

[14] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requir. Eng.*, 15(4):439–458, 2010.

[15] Alexei Lapouchnian and John Mylopoulos. Modeling domain variability in requirements engineering with contexts. In *Conceptual Modeling-ER 2009*, pages 115–130. Springer Berlin Heidelberg, 2009.

[16] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN 9783540741046. URL http://books.google.fr/books?id=anJsH3Dq5BIC.

[17] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. ISBN 0137903952.

[18] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

[19] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[20] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[21] Yogesh S Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In *Theory and Applications of Satisfiability Testing*, pages 360–375. Springer, 2005.

[22] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[23] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3): 59–6, 2010.

[24] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_5. URL http://dx.doi.org/10.1007/978-3-540-73368-3_5.

[25] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. In *AAAI*, 2010.

[26] Peter Mell and Timothy Grance. The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), September 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[27] Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research challenges for enterprise cloud computing. *arXiv preprint arXiv:1001.3257*, 2010.

[28] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *CLOUD 2013: IEEE 6th International Conference on Cloud Computing*, pages 887–894, 2013.

[29] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[30] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997. ISSN 0001-0782. doi: 10.1145/262793.262798. URL http://doi.acm.org/10.1145/262793.262798.

[31] Meta Object Facility. Meta object facility (MOF) 2.0 core specification, 2003. Version 2.

[32] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[33] George Chatzikonstantinou, Michael Athanasopoulos, and Kostas Kontogiannis. Task specification and reasoning in dynamically altered contexts. In *Advanced Information Systems Engineering*, pages 625–639. Springer, 2014.

[34] VítorE. Silva Souza, Alexei Lapouchnian, and John Mylopoulos. System identification for adaptive software systems: A requirements engineering perspective. In Manfred Jeusfeld, Lois Delcambre, and Tok-Wang Ling, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 346–361. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24605-0. doi: 10.1007/978-3-642-24606-7_26. URL http://dx.doi.org/10.1007/978-3-642-24606-7_26.

[35] Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 60–69, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0575-4. doi: 10.1145/1988008.1988018. URL http://doi.acm.org/10.1145/1988008.1988018.

[36] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.175.

[37] Amit K. Chopra, Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Reasoning about agents and protocols via goals and commitments. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 457–464, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9826571-1-9. URL http://dl.acm.org/citation.cfm?id=1838206.1838271.