UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

# Integration of Strategy-oriented and Goal-oriented Approaches to Self-Adaptation

**Carlos Miguel Pereira Bangueses**

**PROJECTO**

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2012

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

# Integration of Strategy-oriented and Goal-oriented Approaches to Self-Adaptation

**Carlos Miguel Pereira Bangueses**

RELATÓRIO FINAL

**PROJECTO**

Trabalho orientado pela Prof. Doutora Maria Antónia Bacelar da Costa Lopes

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2012

# Resumo

O nosso modo de vida moderno é suportado por sistemas de Software complexos dos quais se espera sejam capazes de lidar com aspectos tais como mudanças nos recursos do sistema, nas necessidades do utilizador, ataques de segurança, falhas de Hardware, etc. Muitas vezes, o custo de um destes sistemas deixar de funcionar pode ser muito alto, chegando por vezes a por vidas humanas em risco.

Estes sistemas críticos oferecem muitas vezes adaptações que permitem reagir a estas mudanças, no entanto os custos da adaptação manual destes sistemas tendem a ser muito altos. Ao longo dos últimos anos, a auto-adaptação estabeleceu-se como uma abordagem eficaz para lidar com as mudanças no ambiente em que operam as aplicações. Sistemas auto-adaptativos são capazes de ajustar automaticamente o seu comportamento em tempo de execução em função da sua percepção do ambiente e do próprio sistema.

Existem várias abordagens para a auto-adaptação que seguem o princípio da separação de interesses, i.e., onde a lógica de adaptação é separada da lógica da aplicação. Um objectivo da utilização deste princípio é reduzir a complexidade do sistema. Outro objectivo é a redução dos custos de implementação de um comportamento adaptativo em sistemas existentes.

Uma abordagem para a auto-adaptação deignada por *orientada por estrategias* consiste em considerar que o comportamento adaptativo é fornecido por um operador humano (administrador) na forma de condições e estratégias de adaptação. Esta abordagem tira partido da experiência e conhecimento do operador humano relativamente às situações que este identifica como problemáticas e às estratégias que melhor permitem lidar com elas. Uma desvantagem desta abordagem é que está bastante sujeita a erros humanos, pois nos casos em que o espaço de configurações possíveis é muito grande, encontrar as estratégias de adaptação adequadas é uma tarefa muito difícil. Espera-se do operador humano o conhecimento sobre todas as adaptações e os impactos esperados de cada uma. Outra desvantagem desta abordagem é a incapacidade de reacção a problemas imprevistos.

Outra abordagem diferente designada por *orientada por objectivos* consiste em tirar partido da experiência dos responsaveis pelo desenvolvimento dos diferentes componentes usados no sistema, tal como o seu conhecimento relativamente ao desempenho e qualidades de serviço que os varios componentes oferecem nas suas diferentes configurações.

Nesta abordagem o operador humano que administra o sistema limita-se a fornecer uma política especificada com os objectivos de alto nível do sistema. Essa política descreve o comportamento desejado para o sistema e é usada junto da informação sobre os componentes e suas adaptações para guiar o sistema de auto-adaptação na escolha autónoma das alterações a efectuar. A estratégia de adaptação que será usada para resolver um problema é computada automaticamente pelo sistema em tempo real. Ao contrário da abordagem anterior, esta abordagem não se limita a lidar com um conjunto de situações previsíveis. Quando um problema é detectado, e assumindo que existem adaptações capazes de corrigir esse problema, o sistema é capaz de calcular um plano para o resolver. No entanto o custo da computação do comportamento adaptativo adequado cresce rapidamente com o aumento do espaço de configurações possíveis, o que implica que o comportamento auto-adaptativo desta abordagem pode ter um impacto considerável no sistema. Existe ainda a possibilidade da execução de adaptações incorrectas ou desnecessárias quando a informação dos responsáveis sobre os componentes e suas adaptações não é correcta ou suficientemente precisa.

O objectivo deste projecto é encontrar um meio-termo entre estas duas abordagens, a fim de obter um framework de auto adaptação que pode tirar proveito da experiência e conhecimento humano, enquanto continua sendo capaz de se adaptar autonomamente a situações imprevisíveis. Para isso é preciso isolar os factores comuns e as diferenças entre as duas abordagens.

Este trabalho é feito no contexto do projecto ADAAS "Assuring Dependability in Architecture-based Adaptive Systems" [2]. A abordagem para auto-adaptação orientada por objectivos utilizada neste projecto é a apresentada no trabalho "Self-management of Adaptable Component-based Applications"[24] e foi utilizada como base para este projecto. A abordagem orientada por estratégias utilizada na integração foi inspirada na abordagem descrita em Stitch [7], mas algumas simplificações tiveram de ser feitas.

A arquitectura do sistema foi inspirada na framework Rainbow [8], uma framework que separa a lógica de adaptação e a lógica do sistema alvo em camadas. A camada de adaptação interage com o sistema alvo através de sensores e efetores oferecidos por este. É possível identificar nesta camada 4 componentes responsáveis por cada uma das actividades do modelo MAPE-K [1]:

- O *Model Manager* mantém uma representação do sistema alvo. Esta representação permite analisar o estado sistema. Este componente é responsável pela actividade MAPE-K de monitorização do sistema alvo e consequentemente é o componente que interage com os sensores deste.

- O componente *Architecture Evaluator* é responsáavel pela actividade MAPE-K de análise. Este componente analisa a informação contida no modelo do Model Manager para detectar problemas no comportamento do sistema alvo.

- O *Adaptation Manager* é responsável por planear o comportamento adaptativo adequado para resolver o problema detectado pelo Architecture Evaluator. No contexto desta framework o resultado deste planeamento chama-se estratégia de adaptação.

- O *Strategy Executor* é responsável pela actividade MAPE-K de execução. O componente executa a estratégia planeada no Adaptation Manager recorrendo aos efetores oferecidos pelo sistema alvo.

O sistema resultante foi implementado como um ciclo de controlo a executar numa camada por cima do sistema alvo. O sistema alvo é qualquer sistema composto por varios componentes que podem ser adaptados separadamente. A separação em camadas permite o sistema de adaptação funcionar separado do sistema alvo, monitorizando o desempenho deste, detectando problemas e efectuando alterações no seu comportamento para os corrigir, sem interferir demasiado na implementação do sistema alvo em si. Este sistema de adaptação precisa da seguinte informação:

- As métricas usadas para descrever os vários aspectos do comportamento ou estado dos componentes do sistema. Estas métricas devem descrever também como se pode calcular o estado do sistema a partir dos estados dos seus componentes.

- A política que descreve o comportamento desejado do sistema. Esta política está definida na forma de objectivos que utilizam as métricas explicadas no ponto anterior.

- A descrição dos vários componentes do sistema alvo que podem ser adaptados.

- Os sensores que permitem o sistema de adaptação obter informação sobre o estado dos componentes do sistema alvo. Estes sensores devem oferecer toda a informação necessária para o sistema poder calcular os valores das métricas utilizadas na descrição do estado. Note-se que nem todas as métricas requerem informação proveniente destes sensores, algumas podem referir-se a conceitos abstractos que dependem de aspectos como a configuração do sistema alvo ou dos seus componentes.

- A descrição das adaptações existentes, os requisitos que definem quando podem ser utilizadas e os impactos esperados destas no estado do componente que adaptam. Esta informação é necessária para a abordagem para auto-adaptação orientada por objectivos.

- As estratégias de adaptação capazes de corrigir problemas conhecidos e previstos. Estas estratégias são utilizadas pelo sistema quando este utiliza a abordagem para a auto-adaptação orientada por estratégias.

- As ferramentas que permitem o sistema de adaptação provocar alterações no comportamento dos componentes do sistema alvo, ou seja, os efetores.

O operador humano utiliza as métricas e objectivos de alto nível definidos na política para descrever o comportamento desejado para o sistema alvo. Quando um problema é detectado o sistema de adaptação tenta utilizar a abordagem orientada por estratégias para o resolver. Isto oferece ao operador humano um maior controlo sobre o comportamento adaptativo. Adicionalmente, o sistema tira partido da maior eficiência desta abordagem em resolver problemas conhecidos. No entanto, quando esta abordagem falha quer porque as estratégias existentes são incapazes de resolver o problema ou porque o problema não foi previsto, o sistema de adaptação utiliza a abordagem orientada por objectivos para o resolver. Assim temos um sistema que é capaz de reagir a um problema mesmo quando não existe uma estratégia desenhada para esse efeito por um operador humano.

# Abstract

Our modern way of life is supported by complex software systems that are required to operate under substantial uncertainty, these critical systems are expected to deal with aspects such as changes in system resources, user needs, security attacks, hardware failures, etc.

These critical systems usually offer adaptations that can modify and correct their behaviour, however the costs of manually maintaining them can be very high. Over the past few years, self-adaptation has established itself as an effective approach in dealing with changes in the environment in which many applications operate. Self-adaptive systems are able to adjust their behaviour at run-time in response to their perception of the environment and the system itself.

There are several approaches to self-adaptation that follow the principle of separation of concerns, where the adaptation logic is kept apart from the application logic. This principle helps reduce the complexity and cost of implementing self-adaptation behaviour on the target systems.

One of these approaches depends on a human operator (administrator) to supply the adaptation behaviour through adaptation conditions and strategies. This approach takes advantage of the expertise of the human operator in regards to what a problematic situation is and how best to adapt to it. However, the approach is subject to human error, and in cases where the space of possible configurations is very large finding the right adaptation strategy can be very difficult. Also, this approach can only react to problems that have been predicted by the human operator.

Another approach decentralizes the knowledge over the available adaptations and their results on the system by using information provided by the developers of the various adaptable components. The human operator that manages the system uses high-level policies to configure the desired behaviour for the system. A closed-loop control layer augmented with sensors and effectors uses all this information to compute adaptation strategies at run time. This allows the approach to react to unpredictable problems. However, the computation of an adaptation plan at run time is less efficient the bigger the space of possible configurations gets and can have a considerable impact on the system.

The objective of this project is to merge these two approaches in order to achieve a self-adaptation framework that can take advantage of human expertise while still being able to adapt to unpredictable situations.

**Keywords:** Self-adaptation framework, closed-loop control layer, critical systems

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This chapter introduces the project by presenting the motivation for the development of a way to integrate two approaches to self adaptation. Followed by description of the context in which this work was developed. After that we present the objectives and the contributions of this work.

## 1.1 Motivation

Over the past few years, self-adaptation has established itself as an effective approach in dealing with the changes intrinsic to the environment in which many applications operate and, hence, there has been a growing interest in self-adaptive systems — systems that are able to adjust their behaviour at run-time in response to their perception of the environment and the system itself. However, the development of this type of systems has shown to be more challenging than the development of traditional software systems. As pointed out in a recent research roadmap to Software Engineering for Self-adaptive Systems [11], the proper realization of self-adaptation functionality still remains a significant intellectual challenge because there is a lack of methods, techniques and tools that enable the systematic development of this type of systems.

Some existent approaches that follow the principle of separation of concerns have the adaptation logic defined in terms of a fixed set of adaptation strategies, decided at design-time towards a specific (and implicit) set of goals [8, 7, 18, 15]. These strategies can, for instance, capture the expertise of system administrators that are used to monitor the behaviour of systems and make manual adjustments when they detect problems. In the case of large and complex systems, the space of possible configurations is typically very large and, hence, finding the right adaptation strategy is a difficult and error-prone process. To circumvent this problem, in some cases control theory and/or learning techniques have been used to find the most appropriate system configuration for each situation [9, 3, 31]. This approach has the drawback that it is time consuming and does not deal with unanticipated conditions, for instance, system configurations and workloads different from the ones observed during the learning process.

A different approach to the definition of the adaptation logic is to use high-level policies that capture the desired behaviour of the system [24, 25, 5, 28] and allow the system to automatically select the mechanisms available to adapt the behavior of the base system. This approach requires a specific infrastructure to generate adaptation plans whenever the system is found to be doing poorly. These infrastructures are usually complex and require computation that impacts the overall performance of the system and affects negatively the reaction time. Moreover, they also typically depend on analytical models that, in some cases, are difficult to obtain.

This dissertation focuses on the development of self-adaptive software systems by integrating the two approaches described above, taking advantage of the know-how of the system administrators and the flexibility of automatic run time selection and adaptation. Following a key principle that emerged from previous research [23, 13, 21], this work addresses systems in which the adaptive behaviour results from the introduction of a closed-loop control layer on top of a base system which is augmented with sensors and effectors.

## 1.2 Context

This work was developed in the context of the project ADAAS "Assuring Dependability in Architecture-based Adaptive Systems" [2], a project financed by the programme CMU|Portugal. The project started in November of 2010 and is a research collaboration between the University of Coimbra, University of Lisbon, Instituto Superior Técnico, and Carnegie Mellon University, with Critical Software as the industrial partner.

ADAAS focus on the provision of self-adaptability as a means for achieving dependability and optimise performance, while reducing the systems' development and operational costs. ADAAS objectives includes the development of new languages, techniques and tools for creating adaptive systems that combine flexibility and predictability.

## 1.3 Objectives

The overall goal of this dissertation is to contribute with an approach that facilitates the development of adaptive software systems, providing more flexibility in what concerns the definition of the systems' adaptation logic. Specifically, the aim of this work is to extend the approach presented in [24] in order to support the integration of two complementary ways to guide the adaptation in such systems:

- the definition, at design time, of a portfolio of adaptation strategies that deal with a set of conditions that are anticipated to be problematic for the system;

- the dynamic generation of adaptation plans from high-level objectives for the system, which allows to deal with unanticipated conditions, for which no strategy in the portfolio exists.

For this integration, the concept of adaptation strategy proposed in [7] is adopted, capitalizing on previous work developed in the context of ADAAS.

Another goal is the development of a prototype as a proof of concept for the proposed approach. The prototype is instrumental for the realization of several evaluation experiments in the context of a case study that demonstrate the validity, usefulness and effectiveness of the proposed approach.

## 1.4 Contributions

The contributions of this project are related to the design of adaptive software systems and what mechanisms allow their implementation. They are as follows:

- The implementation of an approach for the development of self-adaptive systems that supports the definition of the adaptation logic both in terms of adaptation strategies and high-level goal policies.
- A prototype supporting the development of self-adaptive systems with the proposed approach.
- A prototype and an experimental evaluation of an adaptable web application built according to the proposed approach.

## 1.5 Document Structure

This document is structured as follows:

- Chapter 2 describes the state of the art. One section explains the Rainbow framework in which design the project is inspired upon. The next section describes Stitch, with a focus on the strategy oriented adaptation logic used for this project. The goal-oriented self-adaptation approach is described next. Finally, a section containing the comparison of the two approaches.
- In chapter 3 we present an overview of how the integration was accomplished. The first section describes how the goal-oriented approach was implemented. The next section describes the implementation of the strategy oriented approach. Finally the integration is explained.
- Chapter 4 presents the prototype that implements the approach.

- Chapter 5 describes the Case Study that was designed to validate and evaluate the approach.
- Finally, chapter 6 contains the conclusions and future work.

# Chapter 2

# State of the Art

The research community has been contributing with several approaches tackling the complexity of designing and constructing self-adaptive software systems. In this chapter, we provide an overview of some approaches that are more important and relevant in the context of the developed work.

## 2.1 Software Engineering for Self-adaptive Systems

Systems that are able to adjust their behaviour in response to their perception of the environment and the system itself have become an important research topic. Recently, the research roadmap presented in [11] attempts to identify the main research challenges for the systematic software engineering of self-adaptive systems, focusing on four essential views of self-adaptation.

**Modelling Dimensions.** There are many aspects of a system that can be taken into account for self-adaptation. The roadmap defines *modelling dimensions* as the particular aspects of a system that are relevant for self-adaptation. These dimensions are grouped in the following four groups:

- System goals — These are the objectives the system under consideration should achieve. They can either refer to the self-adaptability aspects of the application, or to the middleware or infrastructure that supports that application. In this group are dimensions such as evolution (whether the goals can change), flexibility (whether the goals are flexible in the way they are expressed), duration (validity of the goal throughout the systems lifetime), multiplicity (the number of goals associated with the self-adaptability policy) and dependency (relationship between multiple goals for systems with more than one goal).

- Causes of adaptation — These are the changes to the systems context that have to happen to trigger adaptation. The context is information that can be accessed that describes the state of the system and its environment. In this group are dimensions such as source (the origin of the change and can be internal or external to the system), type (nature of the change, it can be functional like a change of purpose, non functional like the need to improve system performance, or technological like a driver update), frequency (how often a change occurs) and anticipation (whether the change can be predicted).

- Mechanisms to achieve adaptation — These capture how the system reacts towards change, how the adaptation is executed. In this group are dimensions such as type (whether the adaptation is related to parameters or structure, or a combination of both), autonomy (degree of outside intervention during adaptation, can the adaptation be executed autonomously or is it assisted), organization (what is the organization of the components responsible for the adaptation, is it a single component or distributed amongst several components), scope (is the adaptation localized or does it involve the whole system), duration (how long the adaptation takes to complete), timeliness (does the adaptation have a guaranteed time period to execute or can that time vary considerably) and triggering (whether the adaptation is initiated by a change triggered event or by a timed event).

- Effects of adaptation on the system — These capture the effects of the adaptation on the system, the result of the execution of an adaptation. In this group are dimensions such as criticality (what the consequences are for a failed self-adaptation), predictability (can the consequences of an adaptation be predicted accurately), overhead (how expensive is the adaptation process itself) and resilience (persistence of service delivery that can justifiably be trusted, when facing changes).

**Requirements.** In order for a self-adaptive system to modify its behaviour as a reaction to changes in its environment, it needs to be able to monitor that environment. The roadmap defines that requirements engineering for self-adaptive systems must address which aspects of the environment need to be monitored, what adaptations are possible and what constrains how those adaptations are realized.

One of the main challenges for self-adaptation is that we cannot anticipate requirements for the entire set of possible environmental conditions. Therefore, the requirements for self-adaptive systems should be able to cope with incomplete information about the environment and the evolution of the requirements at run-time.

**Engineering.** The roadmap strongly advocates that self-adaptive systems should be based on a feedback principle with focus on the control loop when engineering self-adaptive systems. The generic model of the control loop is based on a closed loop with 4 main activities:

- Collect — In this activity information is gathered form the environmental sensors, application requirements and context.
- Analyse — In this activity the information gathered in the previous activity is checked against the Goals of the system.
- Decide — In this activity a decision is made based on the output of the previous activity.
- Act — In this activity the decision that resulted from the previous activity is executed.

These activities coincide with those involved in a four-stage cycle for autonomic computing identified by IBM in [1], known as MAPE-K, and depicted in Figure 2.1.

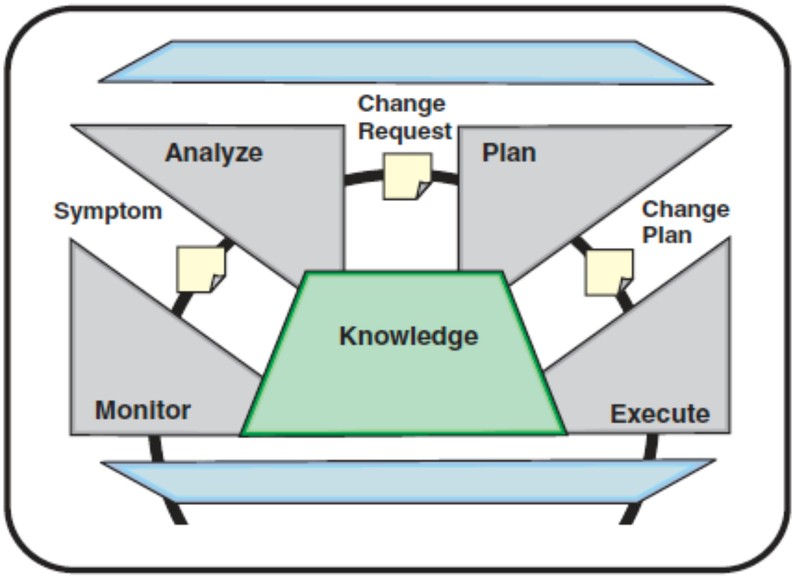

Figure 2.1: MAPE-K model [1].

The control loop is a central element of control theory, which provides well established mathematical models, tools and techniques to analyse system performance, stability, sensitivity or correctness. Recent approaches to control loops in Software Engineering advocate making self-adaptation external, to separate system functionality from self-adaptation.

**Assurances.** Developers need to provide evidence that the set of stated functional and non-functional properties are satisfied during systems operation. However, traditional verification and validation methods rely on stable definitions of software models and properties. Self-adaptive systems support changing goals and requirements and, hence, a verification and validation framework must be supplemented with the run-time assurance techniques [11] such as Dynamic Identification of Changing Requirements (system requirements can change as a result of a change in context), Adaptation-Specific Model-Driven Environments (models allow the application of verification and validation methods during the development process and can support self-adaptation at run-time), Agile Run-Time Assurance (in situations when models that accurately represent the dynamic interaction between system context and state cannot be developed, performing verification activities that address verification at run-time are inevitable) and Liability and Social Aspects (autonomous software adaptation raises new challenges in the legal and social context, for example: if software does not perform as expected, the creator may be held liable).

## 2.2 Rainbow

The Rainbow framework [13, 8] offers a basis for the development of self-adaptive systems based on an independent control loop that monitors, analyses, plans and executes adaptations while keeping the adaptation logic apart from the target system logic and prone to reuse in other systems. It focuses on providing a reusable infrastructure that allows us to build self-adaptive systems with low maintenance and development costs.

The framework uses a component-and-connector architecture model to reflect abstract runtime states of the target system, and an environment model to provide contextual information about the system. In order to get information from the target system into an abstract model, it relies on specific mechanisms to read the target system and understand what is represented in the model.

Figure 2.2 provides an overview of the approach. *Gauges* are the elements that process system-specific information collected by *probes* (sensors). This information is then used to populate the architectural properties of the abstract model that is maintained in the *model manager*. The *architecture evaluator* evaluates the model against architectural constraints to identify problems or potential adaptation opportunities in the system. The *adaptation manager* is triggered by the *architecture evaluator* and uses information about (1) the state of the system contained in the *model manager*, (2) business quality-of-service concerns and (3) a set of adaptations strategies. The *strategy executor* then executes the selected strategy using available *effectors*.

The adaptation support provided by the Rainbow framework can be used by different systems to achieve self-adaptation. This mainly requires the definition of the elements depicted on boundary of the architectural layer and of the translation infrastructure in Figure 2.2. These elements concern the architectural styles involved in the model of the system (purple element), the architectural constraints (pink element) that should trigger adaptation and the adaptation strategies (salmon element). In the next section, we present

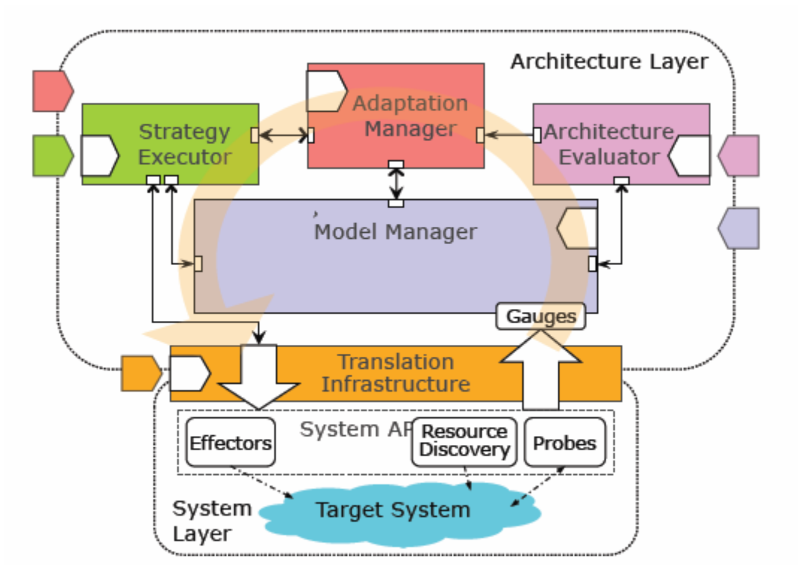

Figure 2.2: Rainbow overview [13].

the language that has been developed by Garlan and colleagues at CMU to specify these different elements.

## 2.3 Stitch

In many systems adaptation is achieved via human intervention. This allows the system to take advantage of human expertise and the ability of experts to understand the anomalies in the system behaviour. A key issue for the autonomic computing approach is whether we can define adaptation strategies that capture what humans do. The development of Stitch language [7] attempts to solve this issue in the context of Rainbow framework.

For expressing adaptation strategies, Stitch defines a language that aims to be sufficiently expressive to represent human expertise and flexible and robust enough to capture complex preferences. More concretely, it allows the definition of self-adaptation operational concepts; support the specification of a value system; allow choice of adaptation and support analysis and composition of multiple independently developed policies.

The key concepts of the Stitch language are described below.

**Architectural style.** The style of the system consists of the component and connector types that can be used in the system as well as a set of architectural constraints. Non-functional properties of component and connector types not directly under the control of the system, such as latency or cpu use, can also be defined. These correspond to the monitored architectural properties. Stitch assumes the architectural style is described in ACME [14], an Architecture Description Language also developed in CMU.

**Quality dimensions and Utility preferences.** Quality dimensions determine what to adapt for and correspond to the business qualities of concern for the target system. Utility preferences define the relative importance between the quality dimensions. Using utility functions it is possible to attribute to each dimension a percentage weight to account of its relative importance.

**Adaptation Conditions.** Adaptation conditions identify conditions for improvement or repair, therefore they determine when to adapt. They consist of architectural constraints and are specified using first-order predicate logic involving certain non-functional properties of component or connector types of the architectural style of the system.

**Operators.** An operator represents a basic command provided by the target system; it corresponds to a system level effector and is a primitive building block for defining tactics. For example, an architectural operator that disables the images on content provided by a given type of component might translate to a system-level effector that changes the configuration of that type of component to use textual mode.

**Tactics.** Tactics provide an abstraction that packages operators into larger units of change, and serves as a logical unit for specifying the cost and benefit of an adaptation step. In Stitch, the tactic is specified as follows:

- It specifies a sequence of operators to be executed.
- It is guarded with a set of conditions that determine its applicability. These conditions are evaluated during run-time with context information to determine whether the tactic can be used.
- It defines a success condition. The success condition is a set of effects that should be observed after the tactic has been completed. These expected effects determine how the tactic execution terminates. The execution of a tactic terminates in success when the expected effects are observed after the tactic application ends, or in error otherwise.
- It specifies its impacts on the quality dimensions of the system.

**Strategies.** Strategies embody the explicit decision choices and provide a packaging construct to constrain adaptation to individual domains of expertise. A strategy encapsulates a set of paths organized in a tree, where each step is the conditional execution of some tactic. This approach needs to be able to update the system state on demand, as it may need intermediate results to choose a tactic at each step. A strategy is defined as follows:

```
module newssite.strategies.example;
import model "ZnnSys.acme" { ZnnSys as M, ZnnFam as T };
import model "ZnnEnv.acme" { ZnnEnv as E };
import lib "newssite.tactics.example";
import op "org.sa.rainbow.stitch.lib.*";  // Model, Set, & Util

define boolean styleApplies = Model.hasType(M, "ClientT") && .."ServerT"..;
define boolean cViolation =
    exists c : T.ClientT in M.components | c.experRespTime > M.MAX_RESPTIME;

strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
  define boolean hiLatency =
    exists conn : T.HttpConnT in M.connectors | conn.latency > M.MAX_LATENCY;
  define boolean hiLoad =
    exists s : T.ServerT in M.components | s.load > M.MAX_UTIL;

  t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
    t1a: (success) -> done ;
  }
  t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t2a: (!hiLoad) -> done ;
    t2b: (!success) -> do [1] t1 ;
  }
  t3: (default) -> fail;
}
```

Figure 2.3: Example of a strategy using two different tactics [7].

- It specifies a tree of condition-action-delay decision nodes. At each node there are a series of conditions that allow a tactic to be chosen. The delay is a time-window before the effects of the tactic can be observed.
- It specifies conditions of applicability; these are used during run-time to choose a strategy that better fits the observed system behavior.

**Strategy selection.** When an adaptation opportunity is detected (i.e., an adaptation condition is detected to hold) and multiple strategies are applicable, the system computes the expected aggregate attribute vector for each strategy. This is done by descending the strategy tree and collecting tactics impacts on cost-benefit attributes as well as the probability defined at the node in the strategy. Finally, the strategy with the highest score is selected.

Figure 2.3 shows an example of a strategy specification in Stitch presented in [7]. The description starts by importing models, or components, that will be referred to throughout the specification of the Strategy. Next, two conditions are defined to be used as applicability conditions. The first, named *styleApplies*, checks whether the current system configuration includes the 2 architectural types used in this strategy. The second condition checks if any of the clients has a response time higher than the defined maximum response time. After the declaration of the strategy *SimpleReduceResponseTime* and its conditions of applicability, two more conditions are defined and are used in the condition-action-delay nodes to guide the choice of edge at each step.

If the conditions of applicability of the strategy are satisfied, and the strategy is selected, then the top level nodes labeled as *t1*, *t2* and *t3* are considered. Once all the conditions of the nodes in the current step have been evaluated, the node to be executed is chosen at random from the set of applicable nodes. After the defined delay time has passed the current system state is updated and the condition of success of the tactic is evaluated. The keyword *success* is a condition that evaluates to true if the current tactic ended in success. The keyword *done* signifies that the objective of the strategy has been achieved, and the strategy terminates successfully. The keyword *fail* signifies that the objective was not achieved and aborts the strategy. The keyword *default* is a condition that evaluates to true if none of the other nodes are applicable, that is, the condition that guards them is not satisfied. If no default branch is defined then one is implied with a fail terminator. The keyword *do* is used to repeat the subtree rooted at the node it points to, the value between brackets ([1] in this example) indicates the number of times the do repetition can occur within a single execution of a strategy.

A strategy can end in success, which means the adaptation objective has been achieved, or in failure. Failures can be registered and integrated in the selection process.

## 2.4 Self-adaptation based on Goal Policies

A different line of research for self-adaptation has developed around the idea of defining the system adaptation logic in a declarative way (instead of operational way) i.e. defining how the system should behave. In these approaches, the planning activity is in general more complex and support for determining the actions required to achieve the goals has to be defined.

The approach proposed in [24] follows this idea and considers that the adaptation logic is defined in terms of goal policies that establishes bounds and target values for key performance indicators (KPIs). This information about the systems expected behaviour is used to trigger events that will initiate self-adaptation, as well as decide on what adaptations should be executed. The approach, as depicted in Figure 2.4, is structured around the following layers:

**Goal Policy Layer.** This layer receives as input the goal policy (which specifies the bounds and target values for the KPIs) as well as the component specification (describing the components used in the system) and the adaptation specification (describing the available adaptations for the components) and generates the adaptation rules that consist of the set of possible adaptations that can be used to correct the system behaviour for each event triggered by the violation of a goal. The approach considers any adaptation that either affects the target KPI positively or doesn't affect it at all as a valid adaptation, this allows adaptations with an effect that varies depending

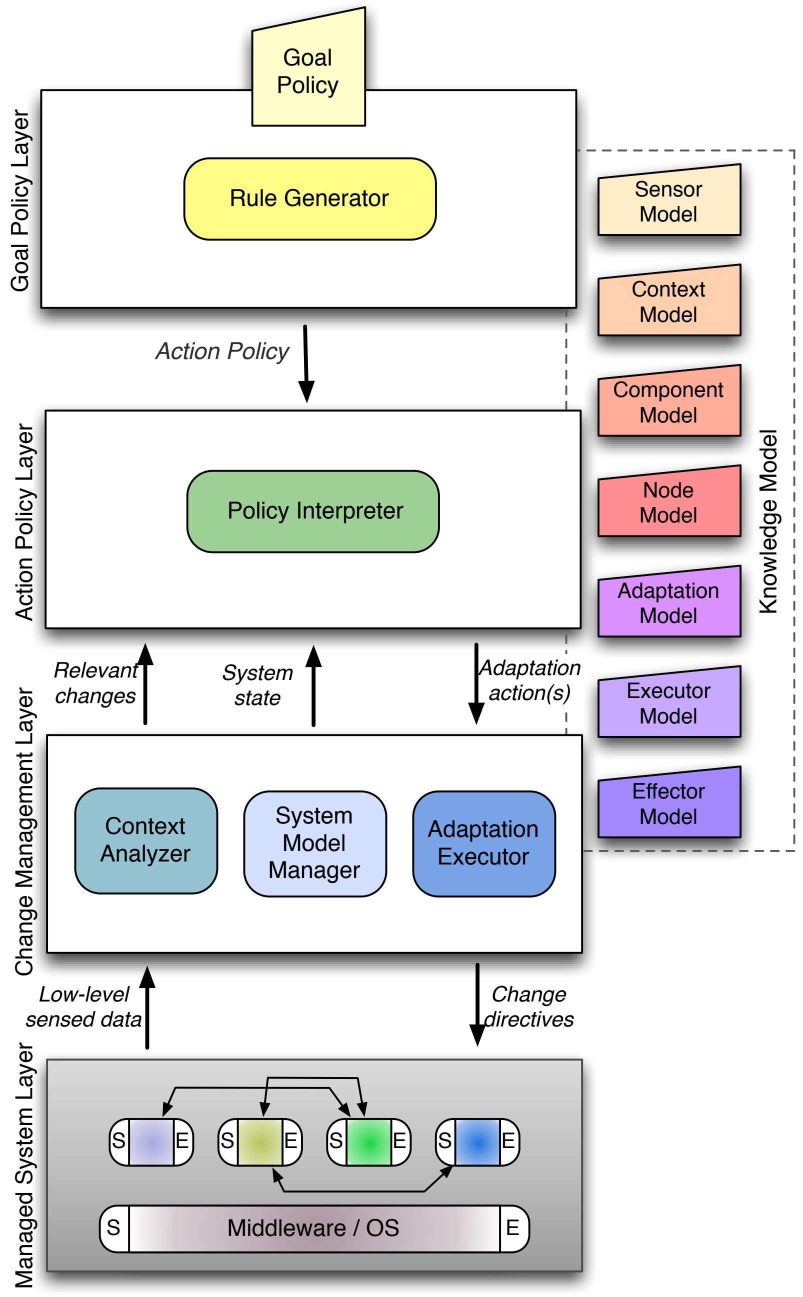

Figure 2.4: Overview of the approach proposed in [24].

on the state of the system to be considered viable adaptations by the Action Policy Layer. The generation of the adaptation rules generally takes place at system deployment time, but can be performed at runtime as a result of a change in goal policy.

**Action Policy Layer.** When a deviation from the desired behaviour is detected, an event is triggered that causes the execution of the policy interpreter in this layer. The policy interpreter evaluates the set of adaptations rules against the system state and the goal policy to choose the adaptations that will, if possible, adapt the system back to its desired behaviour. These adaptations are then sent to the adaptation executor in the Change Management Layer.

**Change Management Layer.** This layer is responsible for monitoring the managed system and triggering the actual adaptation process. The context analyser receives low-level sensed data and detects changes in the system state, it then decides on whether the detected changes are relevant, and if so, triggers an event. After the best adaptations are selected by the Action Policy Layer they are passed down to the adaptation executor that will execute them.

Given that the approach to the development of adaptive systems is an extension of this approach, examples and a more detailed account is provided in the next chapter.

## 2.5 Strategies versus Goal policies

We designate the two type of approaches for self-adaptation embodied by Rainbow&Stitch and the work presented in [24] by respectively, strategy-oriented and goal-oriented. In goal-oriented approaches the system uses its knowledge of all the adaptations available to decide at run-time, according to a the specified system goals, on the set of adaptations that will best correct the observed undesired behaviour whereas in strategy-oriented approaches the system administrator defines a set of adaptation strategies at design time, and the system applies those strategies to correct undesired behaviour.

These two types of approach have advantages and disadvantages that can make one more suitable in one situation and less appropriate in another situation. The key trade-offs of each type of approach are described below.

- Strategy-oriented approach: This type of approach allows us to capture the expertise of the human operator into a set of strategies that are known to solve expected problems. This means that performance of the adaptation behaviour depends on the expertise of the human operators that design the strategies. One advantage is that strategies are defined with applicability conditions which means the selection process at runtime should be fast and efficient. Another advantage is that the adaptation behaviour is predictable for the human operator that manages it. However, designing a strategy requires extensive knowledge of the system and all its components and this can sometimes be unfeasible. The operator that designs a strategy is expected to know all available adaptations and their impact in order to design the best possible strategy. The system is unable to react to unexpected problems in a reliable manner; in some cases an unexpected problem will remain unsolved, in other cases where the symptoms match an existing strategy, this strategy may fail to fix the underlying problem.

- Goal-oriented approach: This approach uses the machines superior computation capabilities to plan the adaptation behaviour at runtime using context information and a detailed description of all existing services and their adaptations. The approach leverages information from component developers concerning the characteristics of each individual component to obtain good approximations of the impact of each adaptation on the component. This means the system takes advantage of the expertise of human operators on a specific component instead of the whole system. Additionally, impacts can be described using information that can only be obtained at runtime. Desired behaviour is encapsulated in a goal policy using high level goals, this goal policy is independent of the available adaptations. However, the complexity of planning adaptation behaviour at runtime increases rapidly the more services and adaptations are available, and the reaction time can become unacceptable. Correct adaptation behaviour depends greatly on the impact functions that describe the expected results of each adaptation. These impact functions can sometimes be very hard to find.

The aim of this work is to integrate ideas from both sides into a single approach in order to attempt to harness the advantages of each of them.

# Chapter 3

# Approach Overview

In this chapter we provide an overview of the approach to the development of self-adaptive systems that is proposed in this project with the aim of supporting the definition of the adaptation logic of systems both in terms of adaptation strategies and high-level goal policies. This approach targets software systems built from several adaptable components, that is, each component may have several implementations and several configuration options. The target system is adapted by changing the implementation or configuration options of one or more components.

The proposed approach integrates a strategy-oriented and a goal-oriented approach to self-adaptation, but can also use each approach in an independent way. More concretely, the proposed approach consists in an extension of the approach proposed in [24] and already overviewed in the previous chapter, in the sequel designated by GRA, through the integration with a strategy-oriented approach developed with that purpose in mind. This strategy-oriented approach, in the sequel designated by SDA, is grounded on the same basic units used in GRA (so the integration becomes possible) and is inspired on Stitch [7]. Both GRA and SDA rely on a closed-loop control layer on top of the base system, with activities organized according to MAPE-K. The most important differences between GRA and SDA are found on the plan and execute stages, as well as in the knowledge they rely on.

In the rest of this chapter, we start by presenting in more detail the several elements of GRA approach already overviewed in Chapter 2, then we present the SDA approach and, at last, we discuss how GRA and SDA are integrated in the proposed approach.

## 3.1 GRA Approach

As mentioned before, this approach targets software systems built by combining adaptable components and relies in high-level goal policies to describe the adaptation logic of the system. As shown in Figure 3.1, the approach can be divided into 3 phases. The design-time phase is when the human operators defines the component specification, adaptation specification and the goal policy. The offline phase is executed before the system is in production. In this phase, given

- a *Components Specification* describing the components used in the system
- an *Adaptation Specification* describing the available components adaptations
- a *Goal Policy* describing the desired system behaviour in terms of a set of *Key Performance Indicators*

a set of *Adaptation Rules* is generated. These rules are then used at run-time, in the so called online phase, to correct undesired system behaviour. The reason to separate the process in an offline phase and an online phase comes from the need to reduce the impact of the adaptation process on the target system. The offline phase executes an analysis of specifications and goals in order to produce a smaller set of possible adaptations to be considered when the system is in production, or online phase. This results in a significant reduction of the adaptations that need to be considered when a problem is detected. The offline phase can be executed any time the goal policy, components specification or adaptations specification changes.

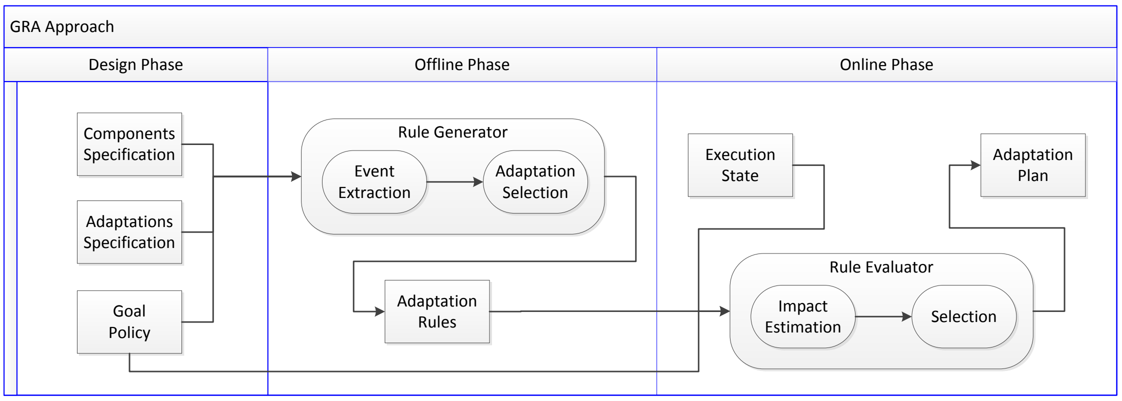

Figure 3.1: Overview of the GRA approach.

In the sequel, the different basic units of GRA are described in detail.

### 3.1.1 Key Performance Indicators

This approach uses Key Performance Indicators to describe specific aspects of the system (System level KPI) or any of its components (Component level KPI), such as CPU or memory use, quality of service, response times, among others. System level KPIs are obtained by applying a combination function on the set of Component level KPI values. These combination functions have to be monotonic, that is, a change on the KPI of an individual component should result in either no change or a change in the same direction on the System level KPI.

KPIs are defined with a name to identify them, the combination function that is used to calculate the System level KPI value, and an error margin. The error margin defines when two values of a KPI are considered equivalent. If the difference between the two values is below the error margin then the two values are considered equivalent. Three examples of KPI definitions are presented below.

```
KPI cpu_use: double  CombFunc Sum  Error 0.01
KPI mem_use: double  CombFunc Sum  Error 0.01
KPI qos: int         CombFunc Sum  Error 0
```

For instance, the first example states that the system level CPU usage KPI can be estimated as the sum of all its components CPU usage KPI values, with an error margin of 0.01.

It is also possible to define a metric though the combination of other metrics. These is achieved through the use of of a combination function that specifies how the values of the different KPIs are combined. These are designated by composite KPIs, or CKPIs for short. As an example consider the definition of the CKPI presented below, which is a measure of the weighted deviation from target CPU and memory usage. The value 0.3 is the target CPU usage while 0.4 is the target for memory use.

```
CKPI ckpi_gdev =
   0.5*|cpu_use-0.3|+0.5*|mem_use-0.4|  Error 0.1
```

### 3.1.2 Goal Policies

The desired behaviour for the target system is specified in terms of a goal policy. The goal policy describes the goals that should be met by the system. The order in which the goals are specified corresponds to the importance of each goal, therefore they are ordered by rank.

The approach identifies two types of goals used to describe the desired behaviour: exact and optimization goals. **Exact goals** are used to define behaviour as acceptable or not acceptable and have 3 subtypes identified below:

```
Goal goal_name : kpi_name Above threshold
Goal goal_name : kpi_name Below threshold
Goal goal_name : kpi_name Between thr_down thr_up
```

*Above goals* state that the desired value of the KPI should be above the threshold; *Below goals* that the value should be below the threshold; *Between goals* that the value should be within the lower and upper thresholds.

**Optimization goals** are used to define an order between the values of a KPI, describing when a value is better than another for a given KPI. Three subtypes of optimization goals are identified below:

```
Goal goal_name : kpi_name Close target
    MinGain mgvalue Every period
Goal goal_name : Minimize kpi_name
    MinGain mgvalue Every period
Goal goal_name : Maximize kpi_name
    MinGain mgvalue Every period
```

*Close goals* state that the desired value for a KPI should be kept as close as possible to the target value; *Minimize goals* that the value of the KPI should be as small as possible; *Maximize goals* that the value of the KPI should be as large as possible. The description of the optimization goals may also include aspects such as the *minimum gain* and the *period*. The minimum gain is used to define when the planned action is worth the cost of adapting the system. The period defines how often the system should attempts to optimize a KPI.

The Goal Policy exemplified below has two exact goals and one optimization goal. The *limit_cpu* exact goal means that the expected CPU use of the target system should stay below 35 percent. The *limit_mem* exact goal also defines a threshold for the amount of memory used by the system. The *max_qos* optimization goal says that the adaptation system should check for opportunities to maximize the quality of service every 20 seconds. The order in which the goals are defined means that the system administrator prioritizes the CPU usage, then the memory usage, and only when both of these goals are guaranteed should the quality of service be improved.

```
Goal limit_cpu : cpu_use Below 0.35
Goal limit_mem : mem_use Below 0.4
Goal max_qos : Maximize qos
    MinGain 0 Every 20000
```

### 3.1.3 Components Specification

The available components and their parameters are defined in the Components Specification. There are two types of components, abstract and concrete. Abstract components represent a group of components that have shared characteristics. Concrete components describe specific components in the target system that have available implementations. Additionally, components can be defined as being subtypes of other components.

The example below is a specification of a concrete component named *Catalog*. This concrete component is a subtype of an abstract component named *StaticContentComponent*. The Catalog component provides static HTML content that contains images and can be configured to use regular quality images, lower quality images or no images at all.

```
Component Catalog
   subtype StaticContentComponent
   Parameters
      ImageQuality: textual, low, regular
```

### 3.1.4 Adaptation

Adaptation behaviour is obtained by either changing the parameters of a component which results in a change in the components behaviour, or changing the configuration of the system which either adds new components, removes existing components or replaces the implementation of a given component. All available adaptations are specified in the Adaptation Specification with the following information:

- The target component of the adaptation. This component is defined in the Components Specification. When the target of an adaptation is an Abstract component then all Concrete components that subtype it are considered targets for the adaptation. Only adaptations that target components that exist in the current system configuration are considered available.
- The set of actions to be performed by this adaptation.
- Requirements that must be met before the Adaptation can be executed. These requirements can refer to aspects such as whether a specific component exists in the current system configuration, the system is in a specific state, or that other adaptation will also be executed.
- Expected impacts of the adaptation on the KPI values of the component it targets described using an impact function. Only the KPIs that are affected by the adaptation have expected impacts. Calculating the impact of an adaptation on a KPI may depend on context information that can only be obtained at run-time, therefore the impact functions are not expected to be completely accurate. However, the system requires that these impact functions describe whether the value of the KPI will increase or decrease. Otherwise it is possible that the adaptations chosen to solve a problem will in fact aggravate the problem. Note that the developer may define multiple adaptations with the same set of actions but different requirements and impacts, this way describing different impacts for the same actions but in different situations.

An example of an adaptation that changes the image quality of the *Catalog* component to low, using a *setParameter(ImageQuality,low)* action, is presented below.

```
   Adaptation ToLowQuality
```

```
Component:
    Catalog
Actions:
    setParameter(ImageQuality,low)
Requires:
    ImageQuality = = regular
Impacts:
    Catalog.cpu_use /= 1.92 // decreases
    Catalog.mem_use /= 1.21 // decreases
    Catalog.qos -= 1 // decrease
```

### 3.1.5 Rule Generation

The goal policy and adaptation specification are used to generate a set of *Adaptation Rules*. This is done during the offline phase without context or run-time information. Adaptation rules consist of an event and one or more sets of adaptations, referred to as adaptation plans:

```
When event
Select {AdaptationPlan1, ..., AdaptationPlanN}
```

Each adaptation plan contains adaptations that can be performed simultaneously and may help achieve the desired behaviour specified in the goal policy. The effects of these adaptations on the system will be evaluated using context information during the online phase, and the total expected effect of an Adaptation Plan will be computed using the combination function of the KPI.

Two types of events can be extracted from the goals in the goal policy. Exact Events are obtained from exact goals and signal the fact that the current state violates the goal. Periodic Events are obtained from optimization goals and are triggered when it is time to check if the system can be improved, according to the optimization goal that originates the event. Events refer to a KPI and describe what kind of impact is considered useful to solve the problem. Consider the limit_cpu goal that describes that the cpu_use KPI should be kept below a 0.35 threshold, this goal will result in an event that should be triggered when the cpu_use is above 0.35 and signals the need to reduce the value of that KPI, that is, it defines positive impact as any impact that reduces the value.

The adaptation plans contained in the rule are generated from the combinations of all compatible adaptations that are considered useful to solve the problem the event signals. Adaptations are incompatible when they have a requirement that defines the other as incompatible. A useful adaptation is an adaptation with an estimated impact that either improves the KPI or has an unknown effect on it. Calculating the estimated impact of an adaptation may not always be possible during the offline phase, as some impact functions may require context information, when this happens the impact of that adaptation is unknown. The reason why adaptations with unknown impacts are considered useful is to ensure no potentially useful adaptation is discarded at the offline phase. The idea behind this process is to reduce the amount of adaptations that need to be considered when a problem is detected at run time.

### 3.1.6 Rule Evaluation

The Rule Generation done in the offline phase results in a set of Adaptation Rules that map each event to multiple adaptation plans, each containing only potentially useful adaptations. The actual benefit of those adaptations can only determined at run time using context information, this is done by the Rule Evaluation process.

The process begins with the removal of all non-applicable adaptations from all available adaptation plans. Adaptations are considered not applicable when the current system configuration does not include the component the adaptation refers to, or any of the requirements of the adaptation is not satisfied by the current system state.

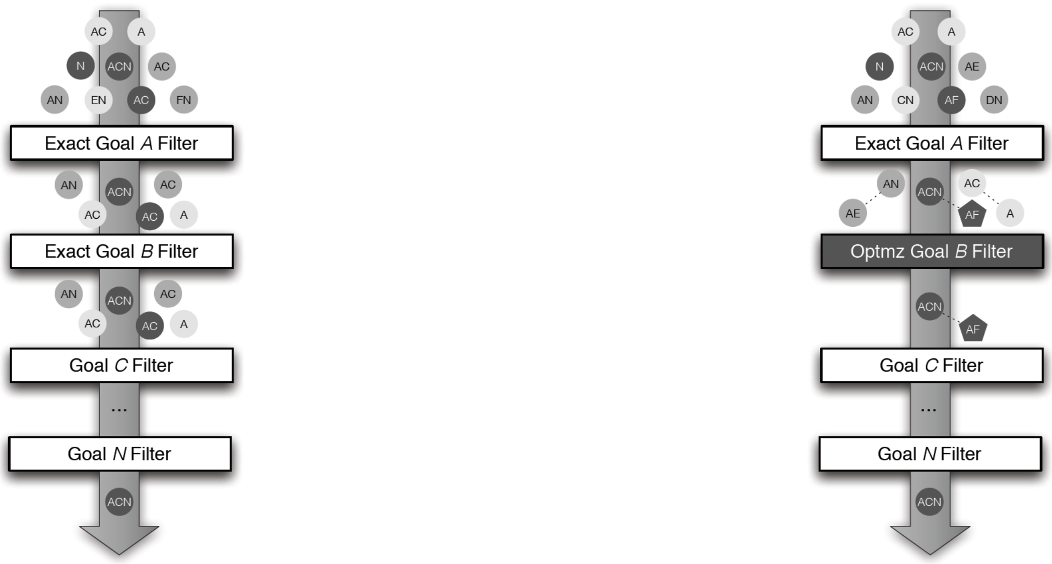

Figure 3.2: Rule Evaluation selection process.

The next step, represented in Figure 3.2, is to use all goals in the goal policy to filter the multiple adaptation plans. For that the system must first calculate the expected impact of each adaptation plan. The expected impact is estimated by applying the combination function defined in the KPI declaration to the current values of that KPI for every component. If the goal refers to a CKPI then the join function defined in the CKPI declaration is applied to the values of its KPIs.

The approach uses Goals as filters using the ranks of the goals to order the filters. This means that an adaptation plan must satisfy the N ranked Goal before it can be considered for the N+1 ranked Goal.

At each Exact Goal filter, only the adaptation plans that are expected to satisfy the Exact Goal of that filter are allowed to pass to the next filter. When no adaptation plans are expected to satisfy the goal then it is considered impossible to satisfy without violating higher ranked goals, as a result the system allows all adaptation plans that reached that filter to pass to the next filter. By doing this we can still satisfy some of the lower ranked goals even though a higher ranked goal will remain in violation.

Optimization goal filters work different since they allow only the adaptation plan that is expected to bring the KPI closer to the target to pass as well as any other equivalent plans. Adaptation Plans are considered equivalent when the difference between their expected impact on the KPI the goal refers to is smaller than the error margin of the KPI.

## 3.2 SDA Approach

The developed strategy-oriented approach relies on adaptation strategies that were inspired on those supported by Stitch [7]. However, several changes were necessary in order to make possible the integration with the GRA approach:

- KPIs used in GRA were used to describe desired behaviour instead of quality dimensions.
- The relative importance of different KPIs is not defined in terms of utility preferences but instead in terms of a goal policy that presents the goals according to its importance (hence, rank-based).
- The type of adaptation actions that can be used in the portfolio of strategies of a system in SDA is exactly the same considered in GRA. As explained in the previous section, this set is determined by the components specification.
- As in GRA, it is the goal policy that defines in which conditions the system is considered to deviate from the correct behaviour (recall that, in Stitch, adaptation conditions were considered for this purpose). More concretely, the approach takes advantage of the offline phase implemented for the GRA to extract the events from the goal policy, and these events signal problems in the system's behaviour.

As shown in Figure 3.3, this approach is also divided into 3 phases. In the design-time phase, human operators have to define a component specification and a goal policy as in the case of the GRA approach. In addition, it must also be provided

- a *Tactics Specification* describing the tactics that can be used by the strategies
- a *Strategy Specification* describing all the defined strategies

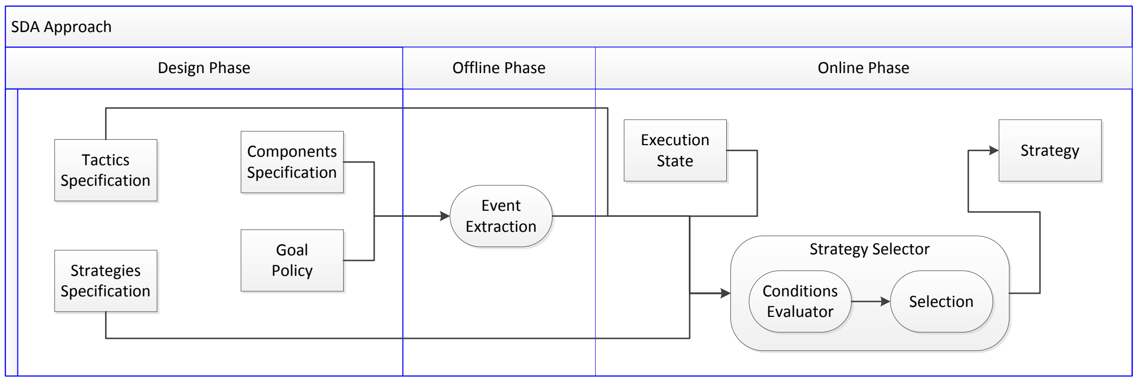

Figure 3.3: Overview of the SDA approach.

In the sequel, we describe in detail the basic units of SDA that were not inherited from GRA.

### 3.2.1 Tactics

As in Stitch, tactics are used to package a set of actions into a larger unit of change. The same primary object *Action* is used by GRA Adaptations and SDA Tactics. Tactics encapsulate the following information:

- The applicability condition that determines whether the tactic can be used. This condition is evaluated at run-time using context information.
- The list of actions to be executed.
- The stabilization period that defines how long the system should wait before any effects of the tactics execution can be observed. The tactic ends when the actions have been executed and the stabilization period has passed.

Unlike Stitch, the tactic does not have a success condition. Instead the condition that describes the effects that should be observed after the tactic ends is specified on edges of strategies. This allows a tactic to have different success conditions depending on the objective of the strategy that uses it. A Tactic is essentially an Adaptation that can affect multiple components and has no defined impacts. This was a design decision made to simplify the approach.

As an example, consider the tactic *decreaseSearch* presented below. The User component is a concrete component that delivers dynamically generated content and has two configurable parameters, Search and Harvest. The Search parameter defines whether the component computes the search results at runtime (timely) or uses cached results (last). The Harvest parameter does the same but for recommendations instead of search results.

The decreaseSearch tactic changes the configuration of the *Search* parameter of all instances of the *User* component from *timely* to *last*.

```
Tactic decreaseSearch
    Actions:
        User.setParameter(Search,last)
    Requires:
        Search = = timely
    Stabilization:
        20 secs
```

The approach includes a special tactic that does not need to be defined by the human operator, referred to as Skip tactic. This tactic has no actions, applicability conditions or stabilization period. The Skip tactic is used only as a tool to simplify the definition of strategies by allowing the creation of edges that do not result in adaptation.

### 3.2.2 Strategies

An adaptation strategy in SDA consists of an applicability condition specifying in which conditions the strategy is applicable (typically, these conditions characterize the problems the strategy may help to overcome) and a Directed Acyclic Graph (DAG) with a root (a node from each all nodes are accessible) describing all the alternative adaptation paths. More concretely, each edge of the DAG is labelled by an applicability condition, a tactic and a success condition. The applicability condition of the edge is used to guide the logic of the strategy at runtime using context information. The success condition is used to determine whether the execution of the tactic was successful or not according to the objectives of the strategy.

The DAG that describes the strategy may include at every node a special edge referred to as default edge. This edge is applicable when all other edges of the current node are not applicable. It is possible to obtain this behaviour without having to implement an exception, by using the negation of the disjunction of the applicability conditions of all the edges of a node. However this edge avoids the re-evaluation of conditions. The corresponding tactic of the default edge is the Skip tactic explained in the previous section, and the success condition is the false condition, which means this edge always ends in failure. This special default edge simplifies the definition of complex strategies. Note that the Strategy Designer has to define the default edge and its target node.

As an example of a strategy, consider the *ReduceCPU* strategy presented below. This strategy is applicable when the system CPU use is over 35 percent and the quality of service of either *Catalog* or *User* components is above 0, that is, it can still be reduced. The strategy attempts to resolve the CPU overload by either reducing the quality of the Catalog component content or the User component content. The human that designed the strategy relies on his expertise to know that configuring the User components to use cached search results or recommendations also reduces the memory use of that component, while reducing or disabling the images of the Catalog component have little effect on any KPI other than the cpu_use. Therefore he designs the strategy to chose to adapt the User component when its memory use is close to the limit defined in the limit_mem goal. Note that the root node has a *default* edge, this edge is used when the memory use of the User component is not above 0.3 but is also not below 0.2, this edge was added to illustrate a possible use for the default edge and allows the strategy to skip to decreasing the Harvest parameter without decreasing the Search parameter. The *fail* condition used in the example is the condition that the previous edge success condition did not evaluate to true, that is, the tactic failed to achieve the objective of the edge it was used in. Figure 3.4 displays the DAG of the ReduceCPU Strategy.

```
Strategy ReduceCPU
    [ cpu_use > 0.35 && (Catalog.qos > 0 || User.qos > 0) ]
    Nodes:
        R, S, H, I, T
    Edges:
        R: [ User.mem_use > 0.3 ] decreaseSearch
            [ cpu_use < 0.35 ] :S
        R: [ User.mem_use <= 0.2 ] decreaseImgQuality
            [ cpu_use < 0.35 ] :I
        R: default :S
        S: [ fail && User.mem_use > 0.3 ] decreaseHarvest
            [ cpu_use < 0.35 ] :H
        S: [ fail && User.mem_use <= 0.3 ] decreaseImgQuality
            [cpu_use < 0.35 ] :I
        I: [ fail ] disableImg
            [ cpu_use < 0.35 ] :T
```

### 3.2.3 Strategy Selection and Execution

The strategy selection process begins when the adaptation system detects undesired behaviour and is divided in two independent steps. Undesired behaviour is identified in the same manner as in the GRA Approach, i.e, by evaluating the current system state against the Goals of the Goal Policy. When a problem is detected an event is triggered and the system needs to select the most appropriate Strategy to address the problem. This is done in the Strategy Selection process.

The first step of the strategy selection process is to evaluate the applicability conditions of all available strategies against the current system state to obtain the list of strategies de-

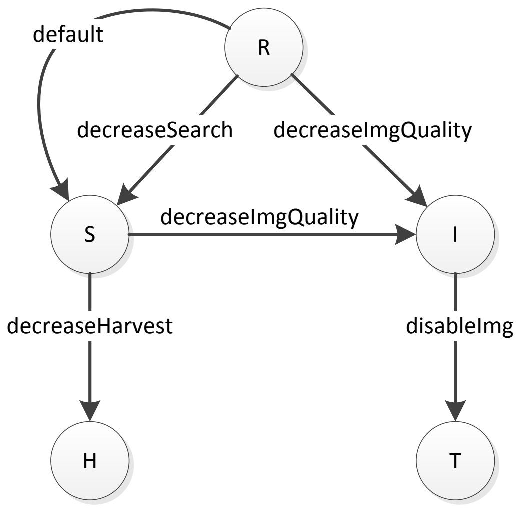

Figure 3.4: DAG of the ReduceCPU Strategy.

signed to address the detected problem. The second step is to apply a selection algorithm to the applicable strategies to obtain the best strategy. In SDA case, the strategy to be executed is selected at random.

Note that Stitch defines impacts on Tactics and probability information on edges in order to compute expected impacts for each Strategy and use this information to select the most appropriate Strategy. As a consequence strategies need to be Trees. The Tactics and edges of the SDA Strategies are not enriched with probability information or expected impacts, therefore the system does not compute the expected aggregate impacts of the strategies during the selection process. Instead it relies only on strategy applicability conditions to determine whether the strategy can be used to solve the problem. This greatly simplifies the definition of Strategies because it does not require the human operator to know the probability information for the edges or the impacts the tactics. Another result of this decision is that we can define Strategies as DAGs instead of Trees. However, it requires the definition of fairly complex applicability conditions to ensure Strategies are used correctly.

The execution process starts at the root node and repeats itself at every node of the DAG until no applicable edges leaving that node are found. The first step is to find all applicable edges of the node. An edge is applicable when the conjunction of the applicability condition that guards it and the applicability condition of its tactic evaluates to true. The first condition is used to guide the logic of the strategy while the second is used to define when the tactic can be used. Once the set of edges that can be taken in the node has been computed, one edge is selected at random from this set and its tactic is executed. When none of the explicitly defined edges is applicable, the default edge is selected if available.

If there is no more applicable edges the strategy has reached its end. The strategy ends in success when the last tactic terminated in success, or failure otherwise.

### 3.2.4 Conditions

All elements of the SDA approach described above use Conditions. Some condition types can be used by all elements, like conditions that refer to the current system configuration or current system state. Others are used by only one element type, like conditions referring to the success or failure of the previous tactic execution. The following condition types have been identified:

- Current system configuration condition. For example, what implementation of a component is being used, or the value of a parameter of a component.
- Current system state condition, described using KPIs. For example the current value of the cpu-use System level KPI, or the current value of the cpu-use KPI for a specific component.
- Condition referring to the result of a previous tactic execution. This condition is defined with an expected result and evaluates to true when the success condition of the last selected edge evaluates to the same value as this expected result. For example, a condition can be defined with the expected result *false* and used to define an edge as applicable only when the previous edge ends in failure.

These conditions can be enriched using the basic Logic operators of Negation, Conjunction and Disjunction.

## 3.3 Integration of GRA and SDA

Figure 3.5 shows the integration of the GRA and SDA approaches. In the online phase there are now two alternative ways to accomplish adaptation. Given that SDA was conceived so it could be easily integrated with GRA, the key element of the integration is the process of decision on when to use one or another at runtime.

The SDA approach takes advantage of the expertise of human operators and offers more control over the adaptation behaviour while the GRA approach takes advantage of the machines greater computation capabilities and knowledge of all available to find solutions to any problem if they exist. The Integrated approach aims to use the strengths of both approaches. This approach is also divided in 3 phases.

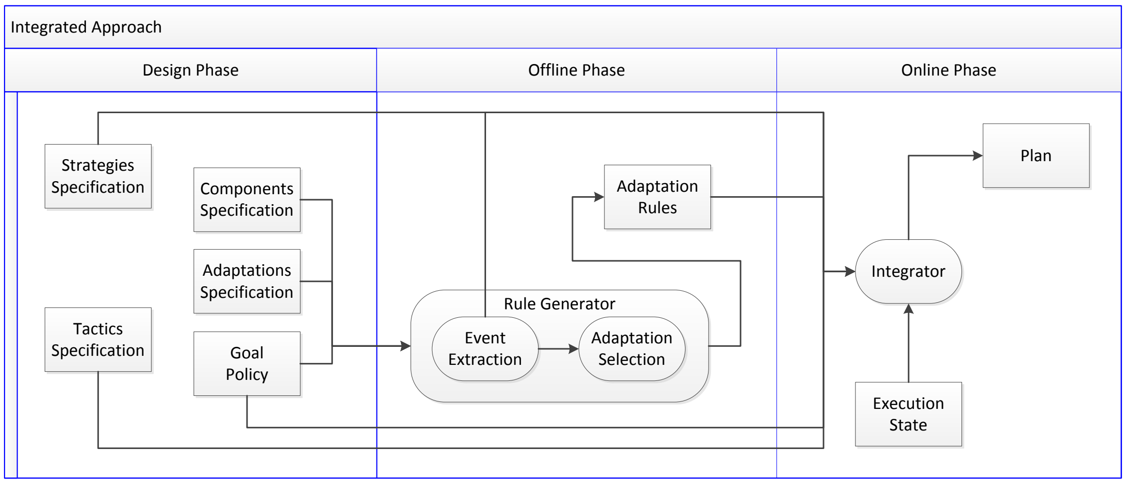

Figure 3.5: Overview of the Integrated approach.

### 3.3.1 Design phase

During the Design phase the information that will guide the adaptation behaviour is defined. The Components Specification and Goal Policy are the same as described in both the SDA and GRA approaches. This means that desired behaviour is described in the same way for the 3 approaches.

The Adaptations Specification of the GRA approach is also included. This contains the information that will be used to generate the Adaptation Rules the Integrated approach relies on to solve unexpected problems, or problems that have no applicable strategies.

The Tactics and Strategies specification of the SDA approach is also included. Note that these specifications can be modified without having to execute the offline phase again. This means the Integrated approach allows the human operator to change the list of available strategies over time.

### 3.3.2 Offline phase

The Offline phase executes the Rule Generation of the GRA approach in order to obtain the Adaptation Rules and the Events that can be used to signal undesired behaviour.

This phase needs to be executed every time the Adaptations Specification, Components Specification or Goal policy changes.

### 3.3.3 Online phase

The Online phase integrator process can use both the GRA approach or the SDA approach to handle an event. Every time a problem is detected this approach will first use all applicable strategies to attempt to solve it. If there are no applicable strategies or all the applicable strategies failed to solve the problem then the Integrator uses the Rule Evaluation process of the GRA approach to obtain an Adaptation Plan that is expected to solve the problem.

The Integrated approach favours human designed Strategies over automatically generated Adaptation Plans. This gives the human operator greater control over the adaptation behaviour for expected problems. Additionally, the human operator may define more than one Strategy to solve a problem because all applicable strategies will be executed before the approach uses the Rule Evaluation process.

Note that the approach will use the Rule Evaluation process to obtain an Adaptation Plan expected to solve a problem that could not be solved using the defined strategies. Therefore the system does not depend on the existence of a strategy for every problem nor that the strategies that exist can actually solve the problem.

# Chapter 4

# Prototype Implementation

This chapter describes the implementation of the prototype that supports the integration of the goal-oriented approach referred to as GRA and the strategy-oriented approach referred to as SDA. In order to accomplish this integration some key components of the system have to be designed to be shared, and the differences should be encapsulated. The system is organized around four main components: the StateManager, the EventLauncher, the EventHandler and the Executor. The main runtime differences between the GRA and SDA approach are managed inside the EventHandler component. Briefly, the responsibilities of these four components are distributed as follows:

- The *StateManager* component maintains a State that describes current system information in terms of the system KPIs. The information contained in this state can refer to the overall system or to any component individually.
- The *EventLauncher* component is responsible for analysing the system State using the Goal Policy in order to detect undesired behaviour or adaptation opportunities. It then signals the situation by launching an event that is captured by the EventHandler and initiates the adaptation behaviour.
- The *EventHandler* is responsible for the adaptation behaviour and encapsulates the main differences between the approaches. In the implementation for the GRA approach this component runs the Rule Evaluation and uses the Executor component to execute each adaptation. In the SDA approach it handles both the Strategy Selection and Strategy Execution, using the Executor to run the tactics. The integration of both approaches is accomplished by using a component that implements the SDA approach to attempt to solve the problem. If this component ends without success the event is passed on to the component that implements the GRA approach.
- The *Executor* is responsible for executing both adaptations and tactics. This component maps the actions of the adaptations and tactics to effectors on the target system.

## 4.1 Dorina Refactoring

The implemntation of the prototype builds upon an existing prototype designed to evaluate the viability of the GRA approach and test its performance, named Dorina. This prototype had to be greatly refactored in order to fit the new design.

### 4.1.1 Dorina overview

Figure 4.1 displays the major components of Dorina.

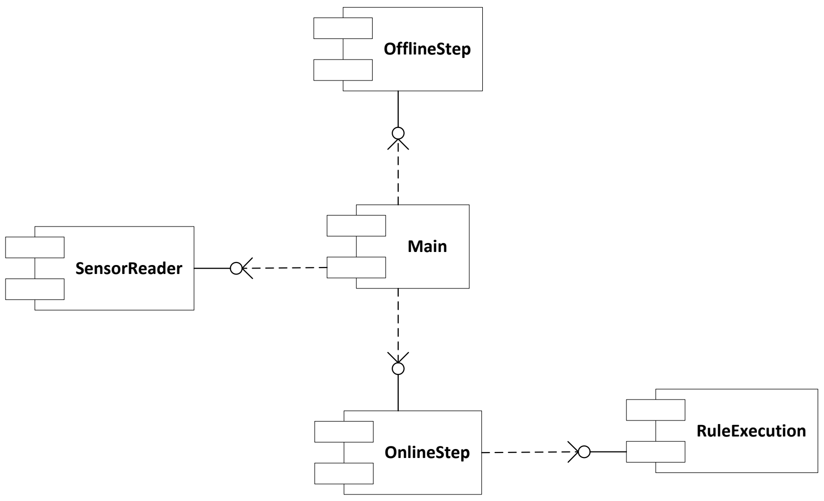

Figure 4.1: Major components of the Dorina prototype.

The component *Main* is responsible for the start up and serves as a bridge between the other 3 components. It loads the definition of the target system Components, all Adaptations and the Goal Policy. In order to prevent confusion between the components of the adaptation system and components of the target system these later ones will be referred to as Services.

The *OfflineStep* component is responsible for all the behaviour described in the GRA offline phase. It extracts the events from the Goal Policy and generates the Adaptation Rules.

The *SensorReader* component is a perl script that enters an infinite cycle and outputs information every minute. It reads various logs in the system to collect the necessary data to describe the state of each service of the system. This script is invoked by the Main component and outputs the information in a line.

When the SensorReader component outputs the line to the Main component this line will be passed to the *OnlineStep* component. All the adaptive behaviour described in the GRA online phase is executed within this component. It starts by translating the line into a usable State object. Then analyses the State against the Goals of the Goal Policy to identify undesired behaviour. If undesired behaviour is detected it will launch an Exact Event, if not it will check if it is time to launch the next Periodic Event. When an event is launched the component executes all the Rule Evaluation logic to obtain an Adaptation Plan.

If an event was launched the resulting Adaptation Plan is passed to the *RuleExecution* component. This component executes all the actions of all the adaptations in the adaptation plan. The execution of an Action depends on the target system and the available effectors.

The whole process is described in figure 4.2 below.

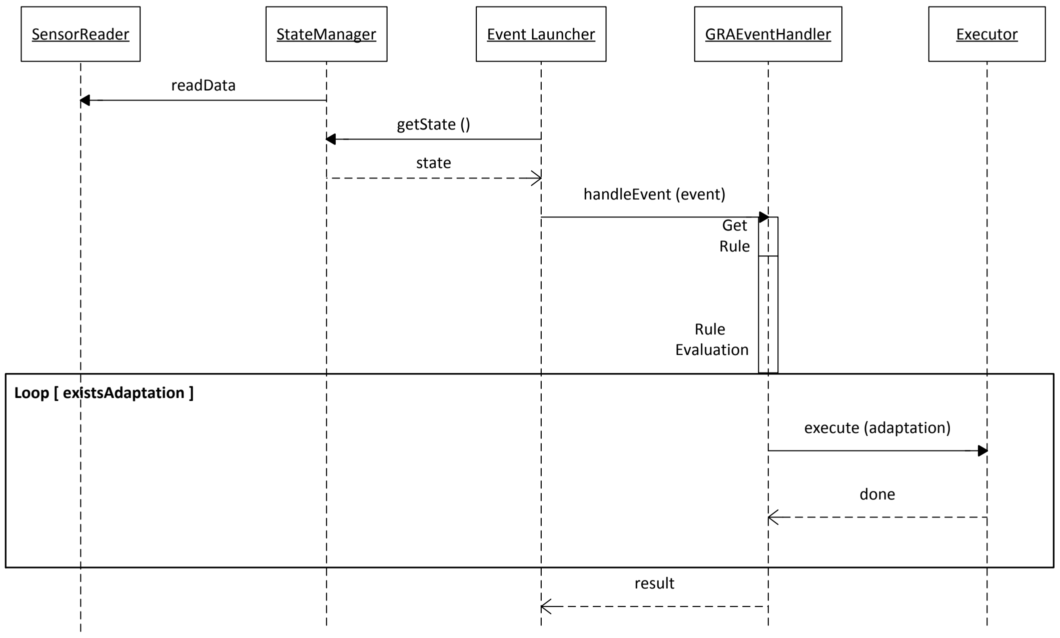

Figure 4.2: Overview of Dorinas execution.

### 4.1.2 New requirements and modifications

The control layer responsible for adaptation runs in the same machine as the target system. This allows the sensors to be accessed easily; however it causes the adaptation system to impact the overall performance of the target system. As a result, optimization is an important requirement for this control layer.

**SensorReader** The original SensorReader reads the logs periodically to find state information referring to one service at a time, then processes all the results into a format that the StateManager can use. However over time the log files become increasingly big and this process becomes heavier. After several attempts to optimize this process one approach was found that results in a very good performance gain. Instead of using the original log files the script starts the process by creating snapshot versions of each log. These snapshots contain only the lines of the logs that happened in the period of time being analysed. Once the snapshots are done the process of the script continues mostly unchanged. This change resulted in a 90% reduction of the time it takes for the SensorReader script to collect the data.

Another change to the SensorReader component came from the fact that the SDA approach requires a new state reading after the execution of every tactic in order to check the success condition. Instead of it running on an infinite cycle with a configurable period it had to be changed to run on demand. Also, the period of time being analysed is no longer constant, instead it is a variable amount of time that depends on the update period of the StateManager and how often Strategies are executed and need State updates. The component was changed to collect all entries in the current minute and save them to the snapshots, except when the current second is below a configurable minimum period threshold then all entries of the previous minute are collected. In this project this minimum period was configured to 10 seconds. For example, when the SensorReader runs at the second 40 of a minute it will collect 40 seconds of data, but if it runs at the second 5 it will discard the last 5 seconds and collect 60 seconds of data of the previous minute. This means that sometimes the effect of an adaptation may be masked and hard to observe seconds after it is executed. A different approach to solve this problem would be to collect the period of time being analysed second by second. This would allow the SensorReader to only analyse fresh data but would also result in a considerable loss of performance. The results obtained from the described implementation are sufficiently accurate for the purpose of this project.

**OfflineStep** In order to encapsulate the differences between the approaches to allow the system to run in GRA mode, SDA mode or Integrated mode, all common behaviour had to be separated. The logic for rule generation and rule evaluation of the original prototype was sound and therefore it needed only to be encapsulated.

The OfflineStep component that was responsible for the Event extraction and Rule Generation had to be refactored to be integrated in the new design. This component is used by all approaches as it produces the Events that are used to signal undesired behaviour, and the Adaptation Rules used by the GRA approach.

**OnlineStep** The OnlineStep component had to be split into multiple components that handle the different activities of the MAPE model. As described before, the StateManager component contains all the logic that involves running the SensorReader script and translating the resulting line into a State object that can be accessed by other components. An EventLauncher component contains all the logic that involves analysing the current state to detect undesired behaviour, if such behaviour is detected an Event is launched. An EventHandler interface with 3 implementations contains all the logic of how the different approaches handle an Event, as well as their integration. The GRA EventHandler implementation encapsulates the Rule Evaluation process of the dorina prototype with some adjustments. The logic of how the Actions of an Adaptation are executed was encapsulated on an Executor component.

Some changes also had to be made to the data structures that support key concepts such as KPIs, Services, Actions, State, among others, and to other minor components, but most of those changes where bug fixes or implementations of missing functionalities. A great deal of effort was spent on finding and correcting bugs that caused errors when the environment and case study of the original prototype changed. Correcting these bugs was costly because they where detected late in the development of the prototype.

## 4.2 Resulting prototype

Figure 4.3 identifies the major components of the system that implement the MAPE activities. The StateManager component is responsible for monitoring the system, maintaining an updated representation of the system state. This is done by using the SensorReader to obtain the data from the sensors. The ExactEventLauncher and PeriodicEventLauncher components analyse the state to find problems. The EventLauncher synchronizes the ExactEventLauncher and the PeriodicEventLauncher, and initiates the adaptation behaviour when a problem is detected. The EventHandler component is responsible for planning the adaptations, and uses the Executer component to execute them.

Figure 4.4 presents a simplified class diagram, with the structure of the code that implements these components as well as data structures. Some of the classes presented below are abstract classes with different implementations that were not included in the diagram. For instance, the Condition class is implemented as AtomicCondition, ConjunctiveCondition, DisjunctiveCondition, NegationCondition, among others.

### 4.2.1 StateManager

The StateManager component is implemented as a thread that processes the data collected by the SensorReader component and uses the result to update the State. This component has a configurable update period that defines how often the State is updated. Smaller update periods will result in more reliable system states, but will also increase the impact of the adaptation system on the target system. Additionally, an update on demand method can be used to force an update of the system state. This is especially useful for the SDA approach, as the Strategy path to be executed is defined by the results observed in the system state after each tactic is executed.

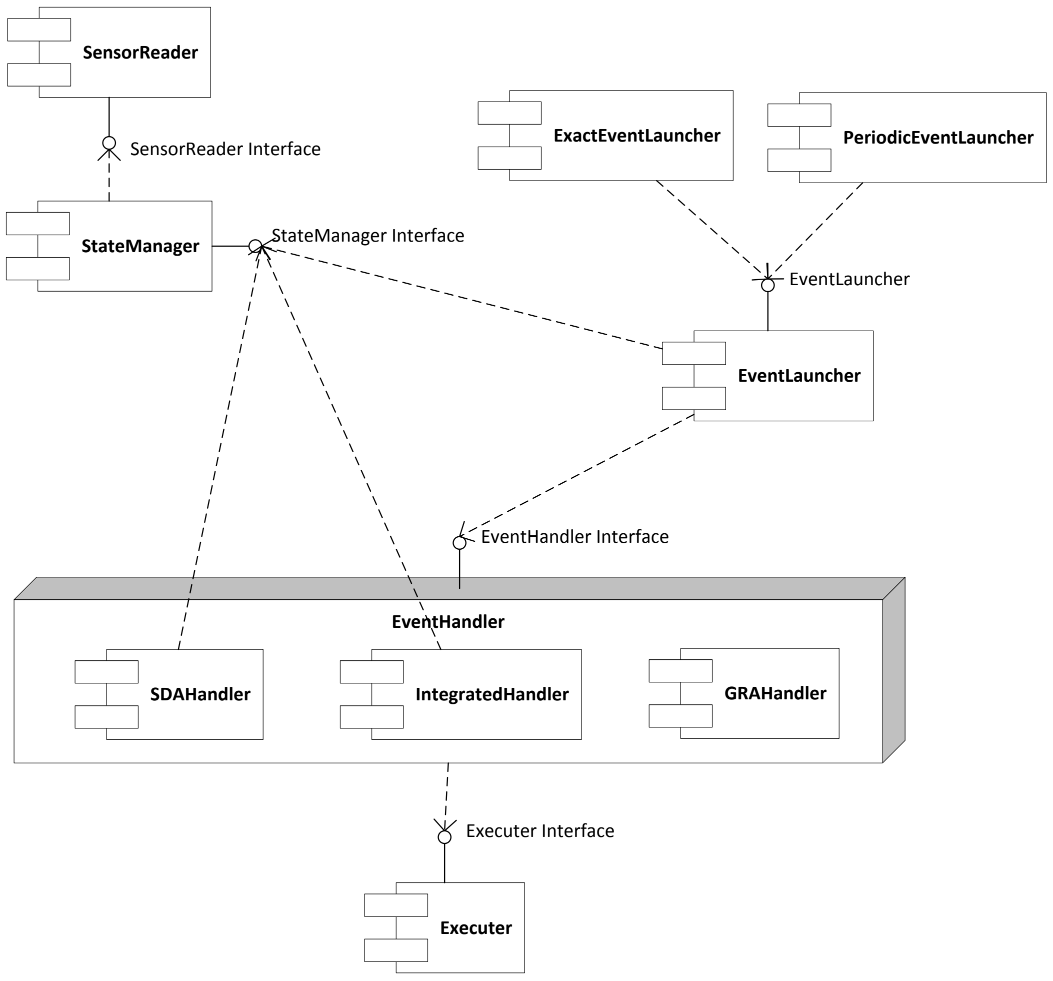

Figure 4.3: Overview of the prototype Architecture.

The SensorReader component reads data from multiple sources and compiles it into a single line containing all the values required to describe a system state. The StateManager component parses the line and translates it into a State that can be used by other components. This happens every time the State is updated. The State is a data structure that contains the values of the KPIs for each component.

### 4.2.2 EvenLauncher

The ExactEventLauncher and PeriodicEventLauncher components are responsible for analysing the current system state to check for undesired behaviour. These components are implemented as threads and have different configurable checking periods, or periods

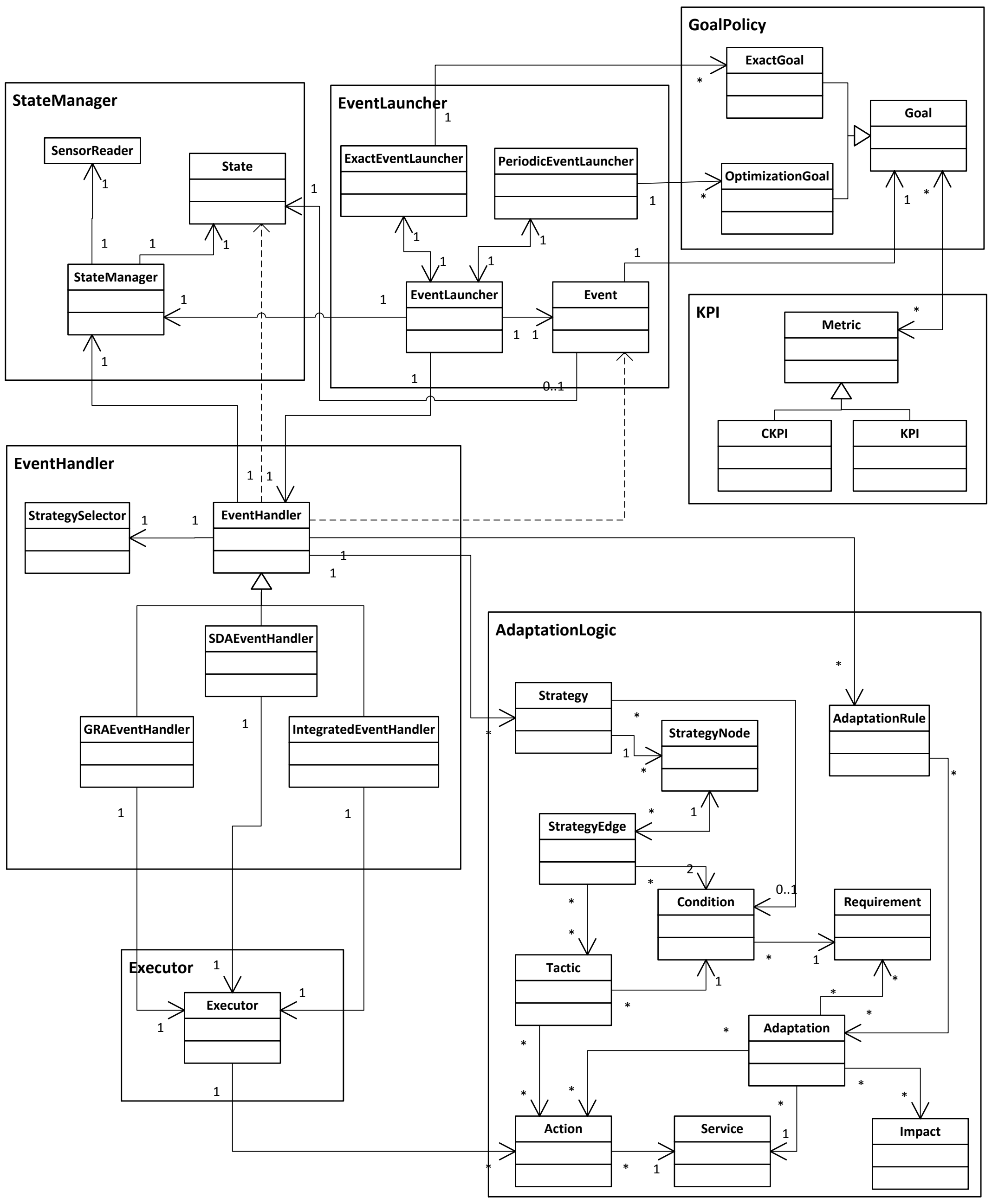

Figure 4.4: Overview of the system classes.

of time between system state analysis. This allows us to configure the system to check for violations of Exact goals and Optimization goals at a different frequency. Exact goals represent acceptable or not acceptable behaviour so they should be checked more often than Optimization goals.

The period of the optimization goal defines when the Periodic Event should be triggered; however in order to guarantee those periods the PeriodicEventLauncher would have to have a very short checking period. Periodic Events can be considered less important because they refer to optimization opportunities, which happen when the system is no longer in distress. It is important to reduce the impact of the PeriodicEventLauncher component on the overall system; therefore it is impossible to guarantee the optimization goal period. Instead, the component uses the period of each optimization Goal to define the order in which the Periodic Events will be triggered.

The EventLauncher component synchronizes the ExactEventLauncher and PeriodicEventLauncher threads, ensuring no event can be launched while another is being handled. If undesired behaviour is detected then the Event associated to the violated Goal is triggered. This means the EventLauncher passes the Event to the EventHandler and enters a blocked state prevent other Events from being triggered.

### 4.2.3 EventHandler

Three different EventHandler component implementations were created to encapsulate the differences in planning and execution of both approaches as well as their integration. These components receive an event as input, plan the adaptations expected to solve the problem that caused the event, and use the Executor component to execute the adaptations on the system.

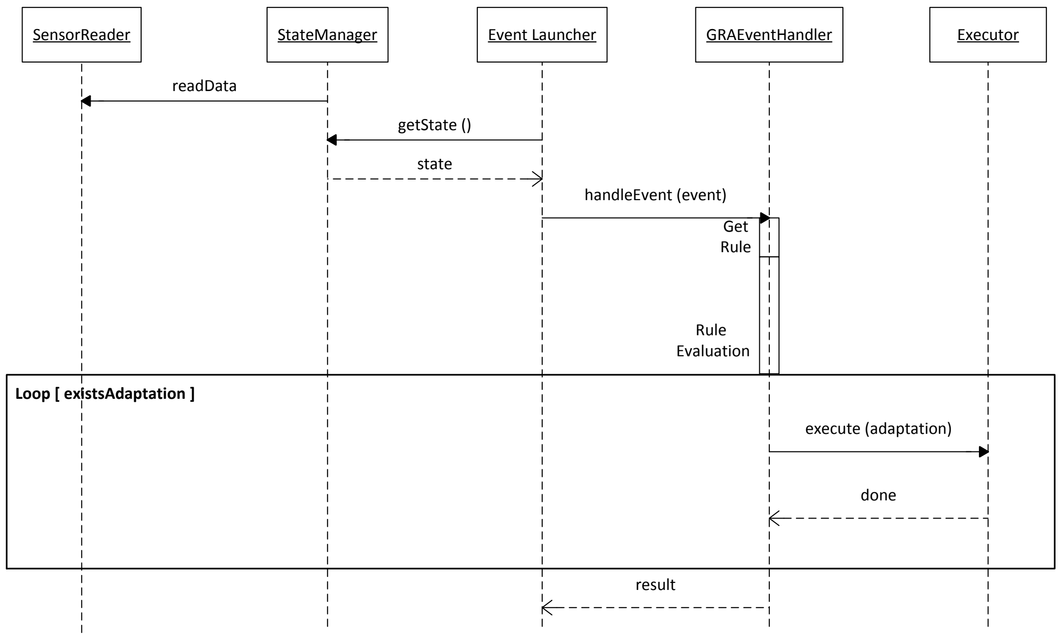

Figure 4.5: Overview of GRA approach execution.

Figure 4.5 displays the sequence of execution of the system using the **GRAEventHandler** implementation. The initial part of the process is the same for all alternatives. The process starts with the StateManager requesting a line of data from the SensorReader.

The next step the EventLauncher gets the current State from the StateManager and analyses it, if a problem is detected an event is passed to the EventHandler, in this case the GRAEventHandler, and the EventLauncher enters a blocked state preventing any new events from being launched. The GRAEventHandler encapsulates the Dorina prototype Rule Evaluation process with some minor modifications to allow it to be encapsulated. Encapsulating the Rule Evaluation logic of the original prototype on a standalone component was not costly but some problems where detected throughout the development and evaluation of this project.

The Rule Evaluation process proceeds as explained in section 3.1. It results in an adaptation plan that should be able to fix the problem that was detected, or atleast bring the system closer to an acceptable state. Once this plan is obtained the GRAEventHandler uses the Executor to execute all adaptations in it. When the adaptations have been executed the GRA EventHandler sleeps for a configurable period, referred to as stabilization period, and then returns, informing the EventLauncher that it finished its process. The EventLauncher will then exit its blocked state and allow new events to be triggered.

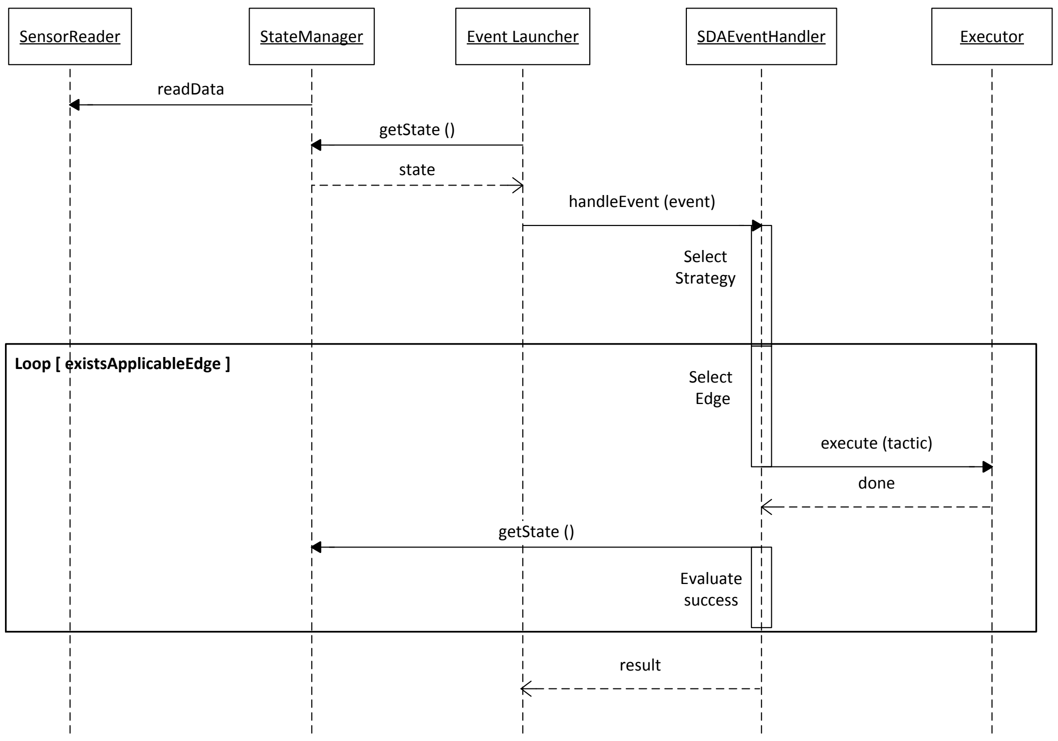

Figure 4.6: Overview of SDA approach execution

Figure 4.6 displays the sequence of execution of the system using the **SDA EventHandler** implementation. As explained before, the first steps of the sequence are the same, the changes happen when the event is passed to the EventHandler. The SDA EventHandler starts by evaluation the applicability conditions of all available strategies against the current system state contained in the event to find the list of applicable Strategies. Then it uses a Strategy Selector component to select what strategies from this list will be executed and in which order. In this projects case two Strategy Selectors where implemented, the first chooses one strategy from the list at random, the second chooses all strategies from the list. The SDA EventHandler uses the first Strategy Selector, but more complex strategy selectors can be implemented. The second step is to execute all the selected strategies in order until one of them ends in success, again, for the SDA EventHandler only one Strategy can be selected, however the Integrated implementation uses the Strategy Selector that selects all applicable strategies. The execution of a Strategy is also implemented in the SDA EventHandler as it involves planning. At each node of the Strategy the SDA EventHandler evaluates the applicability conditions of the edges and their tactics against the current system state to find the list of applicable edges for that node in that context, then selects one applicable edge at random and uses the Executor to execute its tactic. Once the Executor finishes the SDA EventHandler sleeps for an amount of time equal to the tactics stabilization period. Note that unlike in the GRA EventHandler this stabilization period is defined on the Tactic and is not a configurable attribute of the SDA EventHandler itself. Finally, it asks the StateManager for an updated system state and evaluates it against the success condition for that edge. This process finishes when the Strategy reaches a node with no applicable edges, and the SDA EventHandler returns the result of the last success condition evaluation. Just like in the previous case, the EventLauncher will then exit its blocked state and allow new events to be triggered.

The Integrated EventHandler uses the SDA EventHandler and the GRA EventHandler to handle the Event. Figure 4.7 is a sequence diagram for the Integrated approach. When the event is passed to the Integrated EventHandler this event is delegated to the SDA EventHandler. The Strategy Selector used by the SDA EventHandler for this case is the one that selects all applicable strategies. This means that all applicable strategies will be attempted, and the event will only be passed to the GRA EventHandler if all applicable strategies end in failure. The reason for this logic is to make sure that human designed strategies will always be used to attempt to solve known problems, and only when an unexpected problem appears (and therefore no strategy is applicable or applicable strategies do not work) will the GRA approach be used.

### 4.2.4 Executor

The Executor component encapsulates how the adaptive behaviour is accomplished by mapping Actions to system effectors. Both Adaptations and Tactics are ways to group multiple Actions into bigger units of change, and there are no differences on how their Actions are executed. These Actions can be translated into commands when an Adaptation or Tactic is passed to the Executor component to be executed.

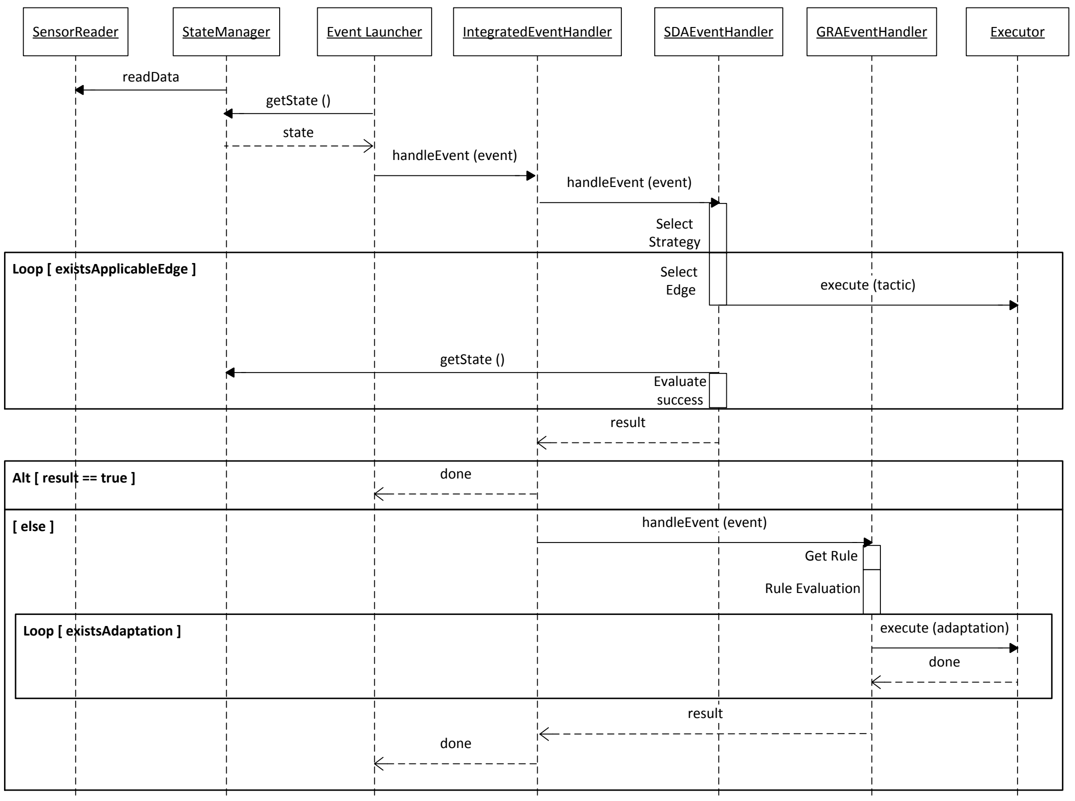

Figure 4.7: Overview of Integrated approach execution

## 4.2.5 Data Structures

The main Data Structures used throughout the project are explained in this section. These classes represent the key aspects of the system:

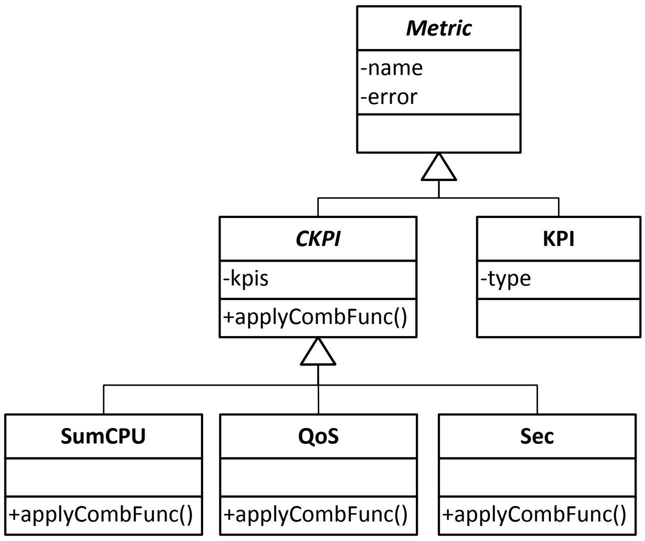

Figure 4.8: Simplified class diagram for KPIs.

**KPIs** KPIs are implemented with a name that identifies them, a type that identifies the combination function required to calculate the System level KPI value, and an error margin. The error value specifies the minimum difference between 2 values before they are considered different. CKPIs have to be implemented, but all share an interface that offers a method used to obtain the value of the CKPI from the State. Note that the State contains the values of all KPIs for each service.

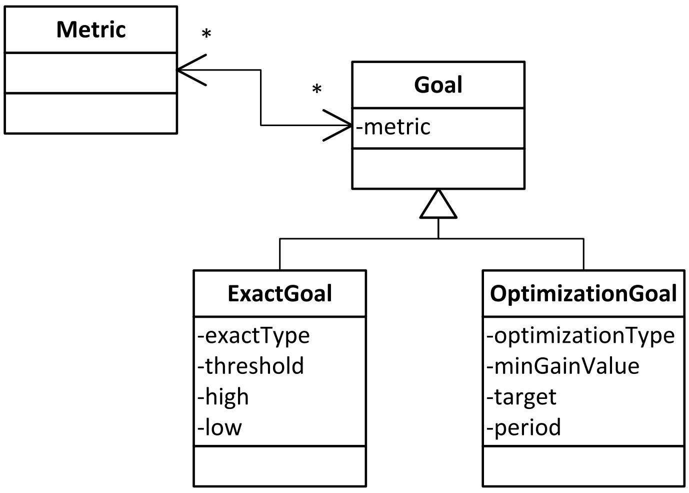

Figure 4.9: Simplified class diagram for Goals.

**Goals** Goals are defined with the metric it refers to, a KPI or cKPI; A minimum gain value that defines when a difference between two values of a KPI is considered worth the cost of adaptation; Finally, a type value that specifies what the goal expects the KPI to behave like. This type can be described as **below**, **above** or **between** for Exact Goals and *close*, **minimize** and **maximize** for Optimization goals.

Exact Goals extend Goals with **high** and **low** values for goals of type between and a **threshold** for goals of types above or below. Optimization Goals extend Goals with a target value and a period.

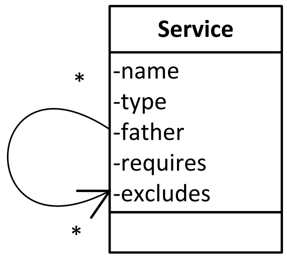

Figure 4.10: Simplified class diagram for Services.

**Services** Services represent the target system components that can be adapted by the adaptation system. The services are defined with the name that identifies the service; A type that allows us to define whether the service is abstract or concrete; If a Service C subtypes another Service P, the father attribute of Service C is a reference to the Service P. Otherwise this attribute is null; If this service requires others services to exist in the current system configuration, these services are contained in the requires attribute; If this service cannot be included in the same configuration as other services, these services are contained in the excludes attribute.

Examples of services can be the Static Premium component that provides static html pages for premium users explained before, the Static Regular component that does the same for regular users, among others. This Service object is used to define what the target of a specific action or adaptation is as well as map each service to its current execution mode or respective KPI values in the State.

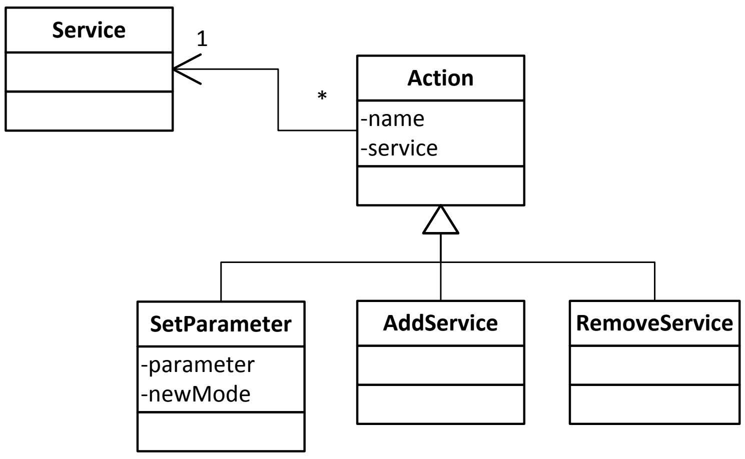

Figure 4.11: Simplified class diagram for Actions.

**Action** An Action is a single command or unit of change that refers to a Service and can be of many types, for example: SetParameter, AddService, RemoveService. Each type has information that describes the action, but as described in the Executor section, the way to execute the action itself is encapsulated in the Executor component. Ideally the actual execution of an action should be in the Action object but this was not implemented due to time constraints. The SetParameter Action changes the configuration of an existing Service, the AddService Action allows us to add a Service to the configuration of the system itself, the RemoveService action does the reverse.

**State** The State describes the overall system state using KPIs and the current system configuration. It offers methods that return the current value of a given KPI for the overall system, as well as for each existing Service. It also offers methods to obtain the current

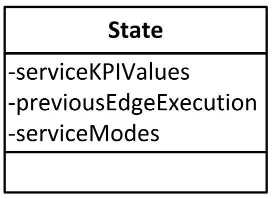

Figure 4.12: Simplified class diagram for State.

execution mode of each Service. The StateManager updates the values contained in the State object using the line returned by the SensorReader component. This line contains information about the average CPU use, medium response time and requests per second for every individual component being monitored. The changes on the current execution mode of a Service is caused by the EventHandler via the StateManager.

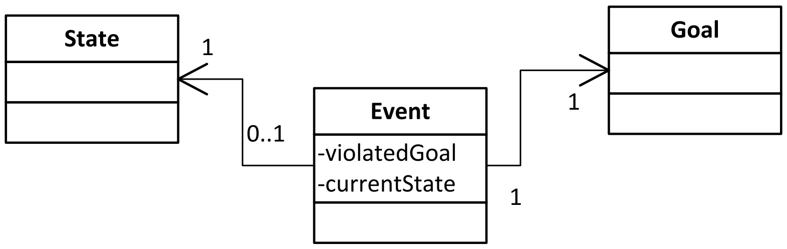

Figure 4.13: Simplified class diagram for Events.

**Event** The Event is the object the EventLauncher creates to signal a problem. This Event contains the current system State and reference to the Goal that is violated. There are 2 types of events, Sporadic events are extracted from exact goals and Periodic events are extracted from optimization goals. Only one Event can exist at any time.

**Requirements** Requirements are used to describe when an Adaptation can be used and can be of 2 types. Metric Requirements refer to the current value of a KPI, they are implemented with a KPI, an operator, a threshold, and an optional service. For example *cpu_use Below* $0.35$ describes a requirement that the current CPU use must be below $0.35$ for the whole system, while *StaticPremium.cpu_use Below* $0.15$ means that the CPU use of the StaticPremium Service must be below $0.15$. Parameter Requirements refer to the current execution configuration of a Service, and are implemented with a Service, a parameter and an expected mode for that parameter. For example, *StaticPremium.ImageQuality low* describes the requirement that the value of the ImageQuality parameter of the StaticPremium service must be low.

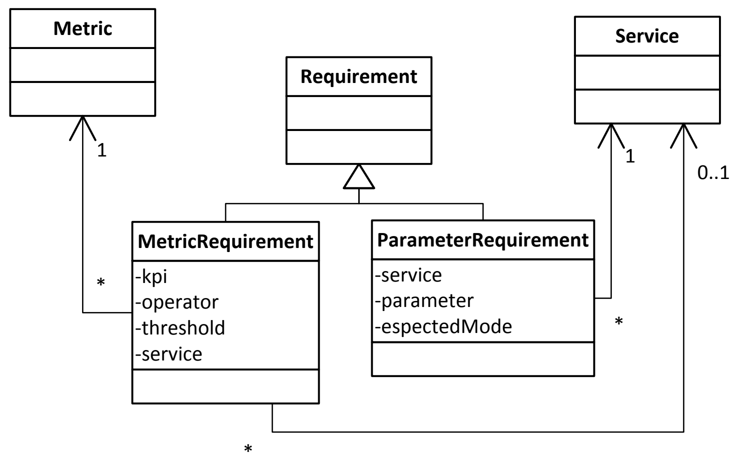

Figure 4.14: Simplified class diagram for Requirements.

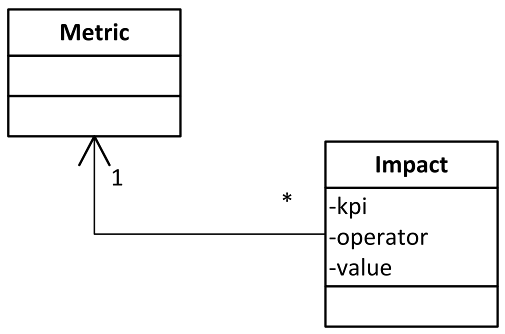

Figure 4.15: Simplified class diagram for Impacts.

**Impacts** Impacts are used to describe the expected impact of an adaptation and are defined with the KPI the refer to, an operator that describes the impact function, and a value. These impacts affect the KPIs values of the service they refer to, not the overall system KPI value. For example, *cpu_use owndiv 2.01* describes that the CPU use of the service the Adaptation refers to is expected to be divided by 2.01.

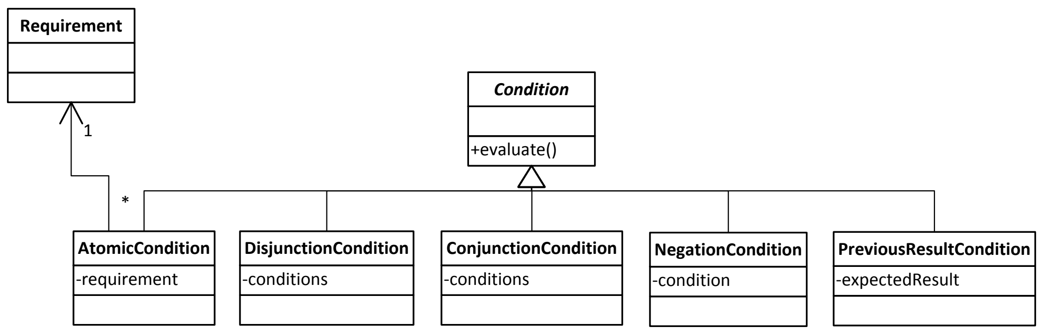

Figure 4.16: Simplified class diagram for Conditions.

**Conditions** Conditions are implemented as described in the DSA section. The Atomic conditions use the Requirements described before to define system configuration or system state related conditions. The previous tactic execution success or failure is saved on the State so that we can define conditions over that information. Logic conditions implemented with other conditions as parameters and behave as described in the DSA section.

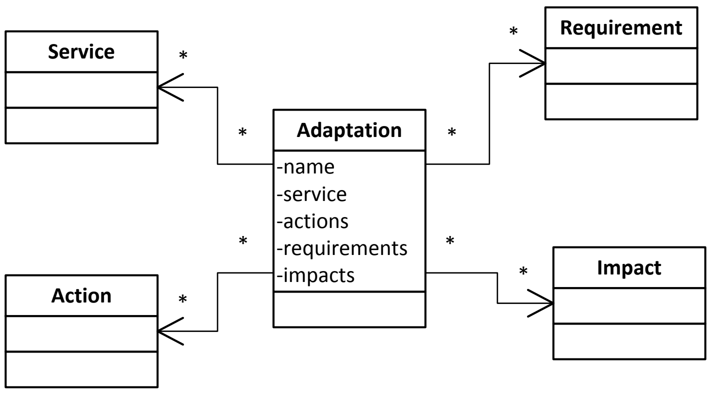

Figure 4.17: Simplified class diagram for Adaptations.

**Adaptations** Adaptations group a set of Actions to be executed that alter the behaviour of a Service. An adaptation has the following attributes: A name that identifies it, the Service it refers to, the set of Actions that must be executed, the Requirements that must be met for the adaptation to be applicable, and the expected Impacts of the adaptation.

The following example is the definition of an adaptation that affects the Static Premium service and has a single action that changes the implementation of the Static Premium service from high quality to low quality. The Static Premium service is the service that provides static html pages with images for premium users. Adapting this service changes the quality of the images. Its only requirement is that the Static Premium service is at high quality, otherwise the action would have no effect. The adaptation has an expected impact on three KPIs. The cpu_use is the average CPU processing power this service is using. The impact on this KPI is *owndiv* meaning it divides the current value of the KPI for this service by the value given in the impact, in this case this adaptation halves the CPU usage. The mrt_premium is the medium response time for premium users. The impact on this KPI is roughly the same as in the cpu_use case. The qos_premium KPI is the quality of service for the premium users. This impact has a *dec* operator meaning it decreases the current value by 1.

```
Adaptation staticPremiumFromMuchToLow =
    new Adaptation("StaticPremiumFromMuchToLow",
        Services.StaticPremium,
        new Action[]setStaticPremiumQualityToLow,
        new Require[]reqStaticPremiumQualityMuch,
        new Impact[]
            new Impact(KPI.cpu_use,Operator.owndiv,2.01),
            new Impact(KPI.qos_premium,Operator.dec,1),
            new Impact(KPI.mrt_premium,Operator.owndiv,1.99)

    );
```

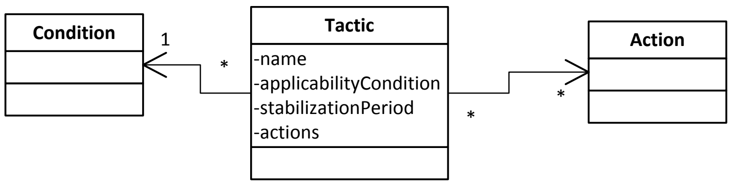

Figure 4.18: Simplified class diagram for Tactics.

**Tactic** Like Adaptations, Tactics also have a set of Actions to be executed. A Condition is used to describe when a Tactic is applicable instead of Requirements, this allows for more complex applicability conditions but may not always be necessary. Because the applicability condition of a tactic is meant to describe in what conditions the tactic can be used, and not to guide its utilization from a strategic point of view, a Require could have been used instead of a Condition. Unlike adaptations where the stabilization period after executing an adaptation plan is a constant value defined in the GRA EventHandler, each Tactic has its own stabilization period.

The following example is the definition of a tactic that does the same as the *staticPremiumFromMuchToLow* adaptation. The StaticPremiumHigh is an Atomic condition defined with the *reqStaticPremiumQualityMuch* requirement. This tactic has a 20 second stabilization period and uses the same Action as the adaptation example.

```
Tactic staticPremiumDecrease =
    new Tactic ("Decrease Static Premium",
        Conditions.StaticPremiumHigh,
        20000,
        new Action[]setStaticPremiumQualityToLow
    );
```

**Strategies** A Strategy is defined with a name that identifies it, an applicability condition, and the root node of a Directed Acyclic Graph where every node has a list of edges and an optional default edge. Each edge has an applicability Condition that is used to guide the execution of the Strategy from a strategic point of view, a Tactic and a success Condition.

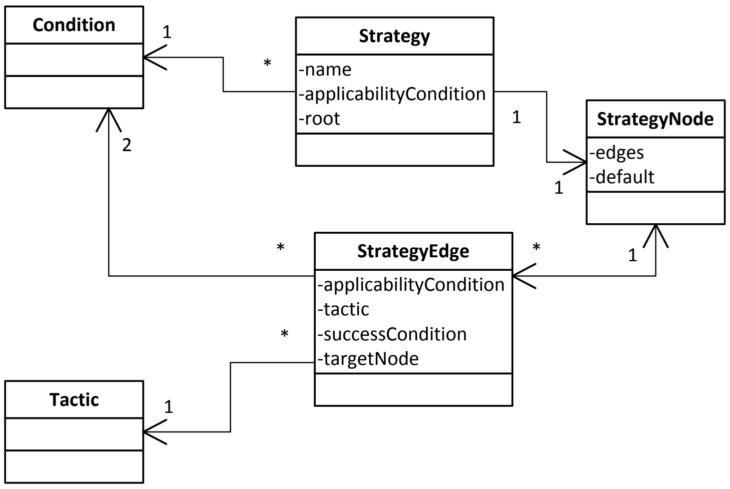

Figure 4.19: Simplified class diagram for Strategies.

```
StrategyNode nodetwo = new StrategyNode();

StrategyNode root = new StrategyNode();
root.addEdge(
    new EdgeLabel(
        Conditions.hiCpu,
        Tactics.staticPremiumDecrease,
    new NegationCondition(hiCpu)
    ),
    nodetwo
);

Strategy twoNodes = new Strategy("Two Nodes",Conditions.hiCpu,root);
```

The example shown above is the definition of a simple strategy with two nodes, named *root* and *nodetwo* and an edge connecting them. The edge has an applicability condition named *hiCpu* that evaluates to true when the system level CPU is above 35 percent, the tactic exemplified in the Tactics paragraph, and a success condition that is the negation of the hiCpu explained before.

# Chapter 5

# Evaluation

The prototype described in the previous chapter was exercised and validated with a case study similar to the one used by the Dorina's prototype [24], that consists of a mock website with 3 types of services that provide Static content, Dynamic content and Secure content. Because the prototype was designed to have 3 execution modes, which allow to run GRA and SDA separately as well as the integrated approach, the case study was also used to evaluate and compare the performance of the three approaches.

The following sections explain the case study in more detail and describe what knowledge the adaptation system was provided with, the experiments designed to evaluate each approach and their integration, and the results of the experiments.

## 5.1 Case Study

As explained before, the target system is a website with three service types, that is, abstract services. Each service type has 2 concrete services, premium and regular. The premium services for each service type provide content for premium users while the regular services provide content for regular users. Essentially, the premium services are heavier than the regular services.

- *Static Content* service type represents components that provide static html pages with images. Regular Static Content and Premium Static Content services belong to this type and are expected to be the most commonly accessed service. These services can be individually configured to deliver content with high definition images, or content with low definition images.
- *Dynamic Content* service type represents components that provide dynamically generated pages. Services of this type have 2 features that can be configured separately: recommendations and search results. Both recommendations and search results can be configured to be generated at runtime or replaced with pre cached results for each client type. The two services of this service type are referred to as *Premium Dynamic Content* and *Regular Dynamic Content*.

- *Secure Content* service type represents components that use the https protocol and other security features. The case study has two services of this service type: *Premium Secure Content* and *Regular Secure Content*. These services can be adapted to deliver high quality images or low quality images much like the Static Content service type.

### 5.1.1 Metrics and Goals

The case study makes use of 9 metrics where 4 of them are KPIs that refer to metrics that can be obtained from the sensors, 6 of them are KPIs that refer to the quality of service for premium or regular users and 2 are cKPIs built using the quality of service KPIs:

```
KPI cpu_use: double  CombFunc Sum  Error 0.01
KPI mrt_premium: double  CombFunc Sum  Error 0.1
KPI mrt_regular: double  CombFunc Sum  Error 0.1
KPI query_load: double  CombFunc Sum  Error 0.1

KPI quality_regular: int CombFunc Sum  Error 0
KPI recomend_regular: int  CombFunc Sum  Error 0
KPI search_regular: int  CombFunc Sum  Error 0
KPI quality_premium: int CombFunc Sum  Error 0
KPI recomend_premium: int  CombFunc Sum  Error 0
KPI search_premium: int  CombFunc Sum  Error 0

CKPI qos_premium =
    quality_premium+recomend_premium+search_premium Error 0
CKPI qos_regular =
    quality_regular+recomend_regular+search_regular Error 0
```

The cpu_use KPI measures how much CPU is used by the target system or any of its components on a scale of 0 to 1. The mrt_premium and mrt_regular KPIs measure the medium response time of the services for premium or regular users in seconds. The query_load KPI measures the requests per second. The other 6 KPIs are used to define the qos_premium and qos_regular CKPIs, that measure the overall Quality of Service (QoS) of premium or regular services.

The expected behaviour described by the Goal policy of this Case Study, ordered by importance from top to bottom, is as follows:

- Goal limit_cpu : The CPU use of the system should be kept under $0.35$ with a $0.01$ error margin.

- Goal limit_mrt_premium : The MRT for premium clients should be under $1.4$ seconds with a $0.01$ error margin.
- Goal maximize_qos_premium : The QoS of the premium clients should be maximized.
- Goal limit_mrt_regular : The MRT for regular clients should be under $1.1$ seconds with a $0.01$ error margin.
- Goal maximize_qos_regular : The QoS of the regular clients should be maximized.

### 5.1.2 Sensors and Effectors

The case study requires the implementation of sensors for the CPU and MRT. As described in the previous chapter, the prototype expects this information to be provided for each service that can be adapted. The Log file that contains the CPU use of each service per request is written by the services themselves. This is a functionality that was imported from the Dorina prototype. However, the command these services used to capture how much CPU time each request used gave us results with values in the order of a 100th of a second. This caused some measurement errors where some requests would use 0 CPU when the CPU had little load. One possible reason for these problems happening on the machine used for this prototype and not on the machine used by the Dorina prototype was that the CPU of this prototypes machine has less cores but each core is faster. The problem was solved by changing the services to use a different command that returns measurements in milliseconds.

The Log file that contains the response times of each request is generated by the Apache server. Using this log we can obtain information on the medium response time for each service as well as the amount of requests per second the service is receiving.

The effectors used in this project are implemented using Linux symbolic links to link the URLs accessed by the clients to different implementations of each service files. Each available Action that adapts a service is mapped to an effector that executes one or more Linux commands saved as Strings using the Runtime Java class. For instance, the Regular Static service has three accessible URLs: *static/regular/file1.html*, *static/regular/file2.html* and *static/regular/file3.html*. Actions that adapt this service link these 3 URLs to versions with high quality images or low quality images.

### 5.1.3 Adaptations

The Static services of the case study have 2 adaptations each, one that increases the quality of the images and one that decreases it. Increasing the quality roughly doubles the CPU use and MRT of the service and increases the QoS. Decreasing does the reverse.

The Secure services also have 2 adaptations each but their impacts are different. Increasing the quality of the service increments its CPU use very little because most of the CPU use of these services is caused by the HTTPS protocol and security functions and not the quality of the content, however the MRT increases greatly, and the QoS increases as normal. Decreasing also does the reverse.

The Dynamic services have 4 adaptations each that configure the search and recommendations to Timely (generated at run time) or Last (uses cached results). These adaptations impact the MRT greatly, but because the search and recommendations are generated on the data server changing these configurations have little or no effect on the CPU use of service running on the Application server.

Tactics using the actions included in the adaptations were also defined. These tactics are used in the strategies explained below.

### 5.1.4 Strategies

The Strategies specification contains multiple strategies but only two are relevant. The non relevant strategies were included to test the strategy selection process. The two relevant strategies are named FixCPU and the RecuperateQoS. The FixCPU strategy is designed to use all available tactics that adapt the Static or Secure services to maintain the System level CPU use under the limit_cpu goal. The RecuperateQoS strategy is designed to reverse the adaptations of the FixCPU strategy in order to improve the quality of service.

Figure 5.1 is the Tree of the FixCPU Strategy. The applicability condition of the FixCPU strategy is that the triggered event refers to the CPU use goal, that is, the limit_cpu goal is violated. This Strategy starts by reducing the quality of the Static service for Regular users (Label A), then the Secure service for Regular users (Label B), then the Static service for Premium users (Label C), and finally the Secure service for Premium users (Label D). If a service is already at low quality when we reach its step the tactic is not applicable but the Strategy should attempt to continue. This means the conditions of applicability for the edge B includes the condition that A is not applicable, for C is that B is not applicable, and so on. The conditions of success of all tactics is that the CPU is no longer overloaded.

One of the reasons the Strategies were implemented as Directed Acyclic Graphs instead of trees was to ease the task of designers that have to define them. The strategy displayed in figure 5.1 is a relatively simple Strategy but results in a complex Tree. The use of DAG allow us to simplify it by using a Default edge with the Skip tactic that does nothing. Using this edge and the fact that we can have multiple paths to the same node we can define a strategy that can skip a step when the tactic is not applicable. Figure 5.2 is a simplification of the FixCPU strategy explained before.

The RecuperateQoS strategy does the reverse of the FixCPU strategy. The applicabil-

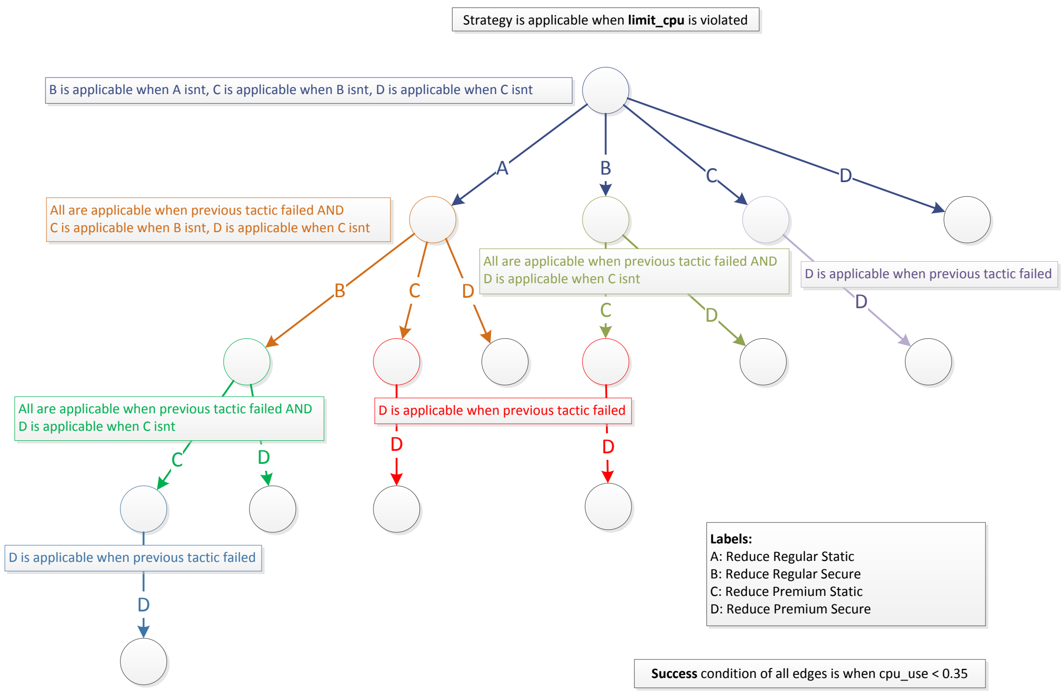
Strategy is applicable when **limit_cpu** is violated
B is applicable when A isnt, C is applicable when B isnt, D is applicable when C isnt
All are applicable when previous tactic failed AND C is applicable when B isnt, D is applicable when C isnt
All are applicable when previous tactic failed AND D is applicable when C isnt
D is applicable when previous tactic failed
All are applicable when previous tactic failed AND D is applicable when C isnt
D is applicable when previous tactic failed
D is applicable when previous tactic failed
A
B
C
D
Labels:
A: Reduce Regular Static
B: Reduce Regular Secure
C: Reduce Premium Static
D: Reduce Premium Secure
**Success** condition of all edges is when cpu_use < 0.35

Figure 5.1: Strategy designed to fix CPU overload displayed as a tree.

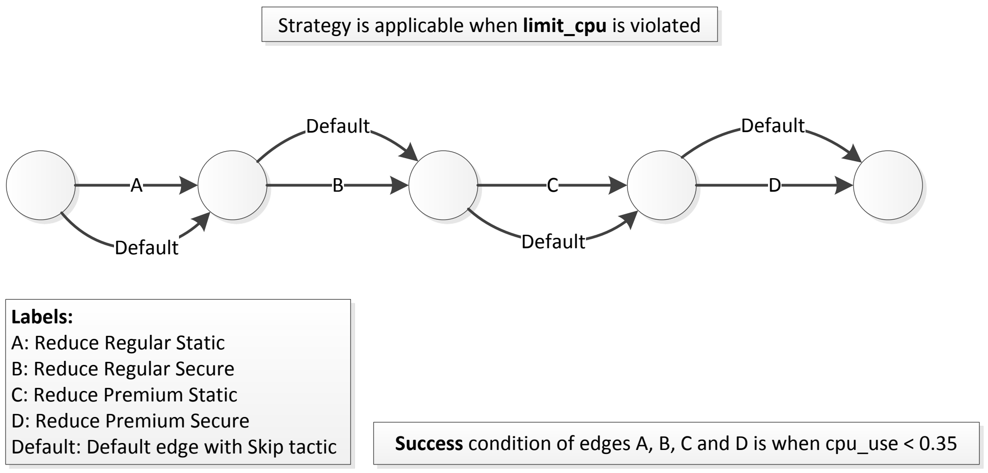
Strategy is applicable when **limit_cpu** is violated
A
B
C
D
Default
Default
Default
Default
Labels:
A: Reduce Regular Static
B: Reduce Regular Secure
C: Reduce Premium Static
D: Reduce Premium Secure
Default: Default edge with Skip tactic
**Success** condition of edges A, B, C and D is when cpu_use < 0.35

Figure 5.2: Strategy designed to fix CPU overload represented as a DAG.

ity condition for this strategy is that the quality of service is not at the maximum level and the cpu_use is below 25 percent. The second part of the condition, referred to as fineCPU, is intended to ensure the strategy does not increase the quality of service when doing so can cause the CPU to go over the threshold. The value of the condition is an estimated value intended to allow at least the first tactic of the strategy to be executed without violating the limit_cpu goal. The success condition of all edges of this strategy is the negation of the fineCPU condition. The applicability condition of all edges includes the fact that the previous tactic was not successful. These two facts ensure the strategy will continue improving the quality of service until the cpu_use goes over 25 percent. The reason for stopping at this threshold is also because the strategy designer knows that the stabilization period defined on the tactics may not be enough for accurate readings and chooses not to risk overloading the CPU.

The order in which each service is improved or reduced in both strategies displays a rank of perceived importance between them. The Premium services are more important than the Regular counterparts. Within both Premium and Regular services, the Secure service types are more important than the Static service types.

### 5.1.5 Test environment

The experimental test bed consists of three machines. The application server machine runs the Apache Web Server and all the components that implement the business logic. The data server runs the recommendation and search engines. The Client machine runs workload generators using Pylot, functioning as clients. The machines are connected by a 100 Mbps Ethernet.

The application server is a Quad core Intel(R) Xeon(R) CPU X3370 running at 3.00GHz with 8GB RAM. The operating system is Linux (Kernel 2.6.32-5-amd64) and the web server is Apache HTTP Server v2.2.21. The Apache HTTP Server is configured with 150 MaxClients and a KeepAliveTimeout of 15 seconds, with CGI, SSL, and rewriting modules enabled.

The Client machine is a Dual core AMD CPU running at 2.00GHz with 4GB RAM. The operating system is Linux (Kernel 2.6.32-5-openvz-amd64). The workload is generated by Pylot (http://www.pylot.org), an open source tool written in python for testing performance and scalability of web services. The tests are based on an XML file that describes the workload. The tool allows us to configure the number of clients and the interval between requests.

## 5.2 Experiments

Two test scenarios were designed and tested with the adaptation system configured to use the GRA event handler, the SDA event handler and the Integrated handler. These scenarios are referred to as CPU Overload and MRT Overload. The objective of the case study is to illustrate the advantages of the integrated approach, for that purpose one of the test scenarios will cause a problem that is considered expected and the other will cause a problem that is considered unexpected. This means a Strategy exists that is designed to solve the expected problem, but a plan will have to be created at run-time by the GRA approach to solve the unexpected one.

The CPU Overload scenario is designed to overload the CPU of the target system by flooding the target system with requests. This causes a violation of the limit_cpu goal described before, and is a problem that is considered expected. The MRT Overload scenario is designed to cause an increase in the medium response time, referred to as MRT, without overloading the CPU of the target system. This causes a violation of both limit_mrt_premium and limit_mrt_regular goals and is considered an unexpected problem. The scenario takes advantage of the fact that Dynamic services have to query an external recommendation and search engine and wait for the response. An overload on the external search and recommendation engine will affect the target systems response times without increasing its CPU load.

The Pylot client program that was used to execute the requests allows us to configure 3 different workloads. All workloads have the same number of simultaneous clients and duration, but the period between requests of each workload is configured individually. Once configured, the Pylot client will start with the first workload, then the second, then the third, and finally it will execute the first workload again.

### 5.2.1 CPU Overload Scenario

The CPU Overload scenario takes advantage of the Static and Secure service types to cause the System level CPU use to go over the limit_cpu goal threshold. The GRA approach should generate an Adaptation Plan that uses adaptations that affect these services to correct the problem. The SDA approach should use the FixCPU Strategy to obtain similar results. The Integrated approach should not need to rely on the GRA approach because the FixCPU Strategy covers all the useful adaptations.

The Dynamic service types have little to no impact on the CPU because most of the processing is done on the Data server, and the adaptation system is targeting the Application server. Adaptations that affect the Dynamic service have been defined with no impact on the CPU in the Adaptations Specification and should not be used by this scenario.

| Service | Clients | Light | Medium | High |
|---|---|---|---|---|
| Static | 30 | 900 | 450 | 450 |
| Dynamic | 30 | 7000 | 7000 | 7000 |
| Secure | 30 | 7000 | 7000 | 1300 |

Table 5.1: Configuration of Pylot for the CPU Overload scenario.

Table 5.1 shows the configuration of the Pylot client for this scenario. The first workload, referred to as Light, should be light enough that the target system can handle it without requiring adaptation, this load is useful to ensure the system does not adapt unless needed and that the system can recuperate the quality of service while under a comfortable workload. The second workload, referred to as Medium, increases the frequency of the requests to the Static service types and should bring the CPU use over the threshold causing the system to adapt. This workload should not cause a problem that requires all available adaptations. All approaches should adapt only as much as necessary. The third workload, referred to as High, increases the frequency of the requests to the Secure service types and should increase the CPU use again. This time the system should use all the adaptations that are still available to attempt to solve the problem. As explained before, the first workload is then repeated. The first time this workload is used it should be observable that the system can handle it comfortably. Therefore, the quality of service of the system should be recuperated during this workload.

### 5.2.2 MRT Overload Scenario

The MRT Overload scenario uses the fact that the Dynamic service types depend on a third system for results to increase the medium response time without affecting the CPU use too much. As the load on these services increases they should have to wait longer for responses from the Data server, thus increasing the time it takes for them to respond without increasing the CPU they use. The GRA approach should be able to solve the problem by adapting the services to use cached results instead of real time generated ones. The SDA approach however has no strategy to resolve the issue, and the goals will remain violated. The Integrated approach should first try to used the SDA approach to solve the problem, fail and then use the GRA approach.

The Static and Secure service types have been defined with impacts on the MRT, however the impact of these services on the overall value is very small.

| Service | Clients | Light | High | High |
|---|---|---|---|---|
| Static | 30 | 900 | 900 | 900 |
| Dynamic | 30 | 7000 | 3000 | 3000 |
| Secure | 30 | 7000 | 7000 | 700 |

Table 5.2: Configuration of Pylot for the MRT Overload scenario.

Table 5.2 shows the configuration of the Pylot client for the MRT Overload scenario. This scenario has only two workload levels, Light and High. To accomplish this the High workload was repeated instead of using a Medium workload. The Light workload is the same as in the CPU Overload scenario, and should not cause any adaptation. The objective of this workload is the same as in the previous scenario, to test if the system will recuperate under a comfortable workload. The High workload increases the frequency of the requests to the Dynamic service types. This will cause a considerable increase in the

MRT. The increase of the medium response time is accomplished by overloading the Data Server, a problem that is considered unexpected.

## 5.3 Results

The results, graphically presented in the next subsections, were obtained from execution logs produced by the prototype and the Pylot program. The graphics were annotated with information like goal thresholds, workload changes, when the event was launched and what adaptations were executed. The following sections present the results of each scenario for each of the three approaches.

### 5.3.1 CPU Overload Scenario Results

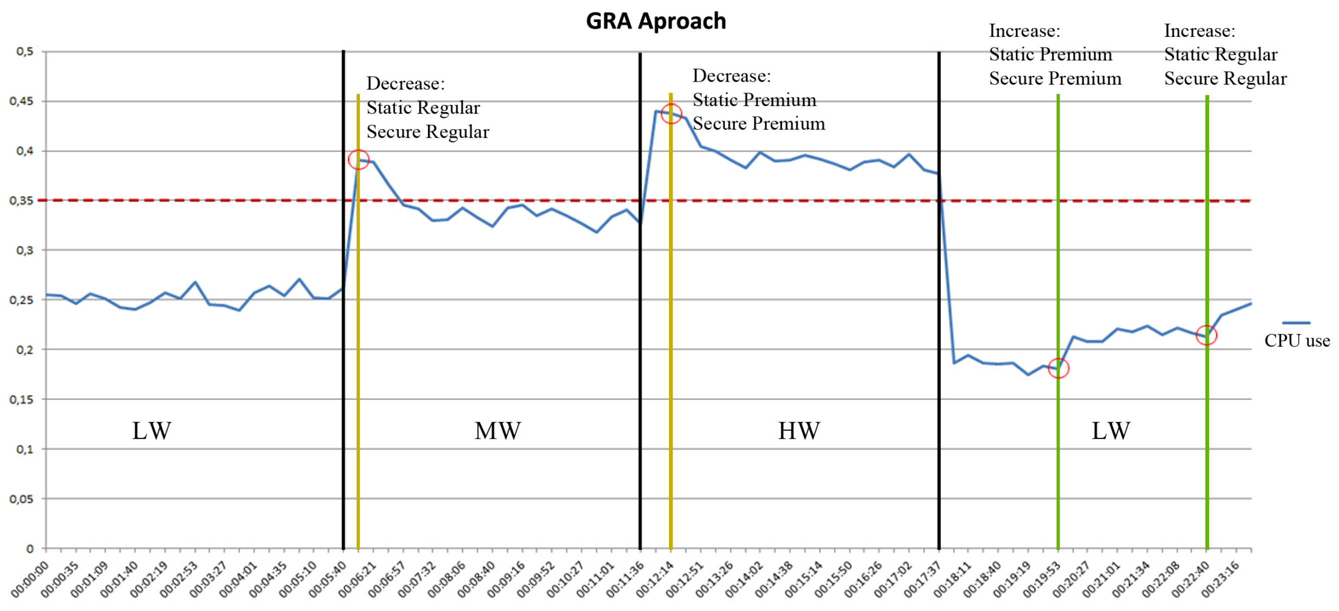

Figure 5.3: Results of the CPU Overload scenario on the GRA Approach.

Figure 5.3 displays the results of the CPU Overload scenario with the adaptation system using only the GRA approach.

We can see that during the Light workload (LW) the overall CPU use value remains around the 25 percent. The goal threshold marked as a red dashed line is at 35 percent, so the adaptation system does not have act during this workload.

Once the Medium workload (MW) begins the CPU use immediately increases and passes the threshold, the event is launched and the system chooses to adapt the Regular services of both Static and Secure service types. This happens because the order in which the two maximize quality of service goals are defined describes that the quality of service of the Premium services is more important. However, if the estimated impact of adapting these two services was not considered enough to fix the problem then a Premium service might have been adapted instead.

When the High workload (HW) begins the CPU use goes over the threshold again. The system uses all the remaining adaptations that impact CPU use to attempt to solve the problem.

Finally, the Light workload is repeated. Because the PeriodicEventLauncher has a much bigger checking period it takes the system some time to detect that the quality of service can be improved. The reason why only the premium services are improved first is because the periodic event that was launched at that time is the one that refers to the maximixe_qos_premium goal.

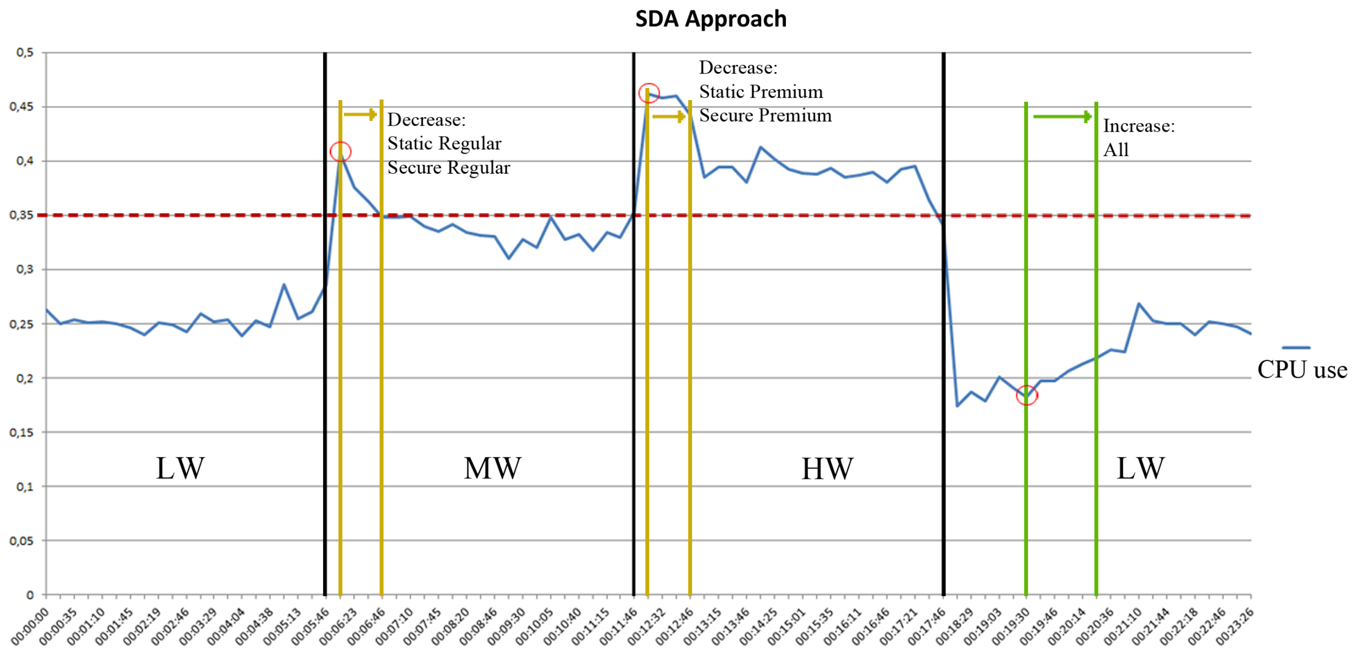

Figure 5.4: Results of the CPU Overload scenario on the SDA Approach.

Figure 5.4 displays the results of the CPU Overload scenario with the adaptation system using only the SDA approach. The system displays very similar behaviour as the one observed in the previous test. One major difference is that the system fully recuperates the quality of service in a single strategy during the last workload. The GRA approach recuperates either Premium or Regular services on each periodic event, therefore it requires 2 periodic events to fully recuperate the QoS. This happens because the RecuperateQoS Strategy does not take into account whether the event refers to Premium services or Regular services.

Figure 5.5 displays the results of the CPU Overload scenario with the adaptation system using the Integrated approach. The behaviour observed in this test is very similar to the one observed in the other tests. The FixCPU strategy includes all known adaptations

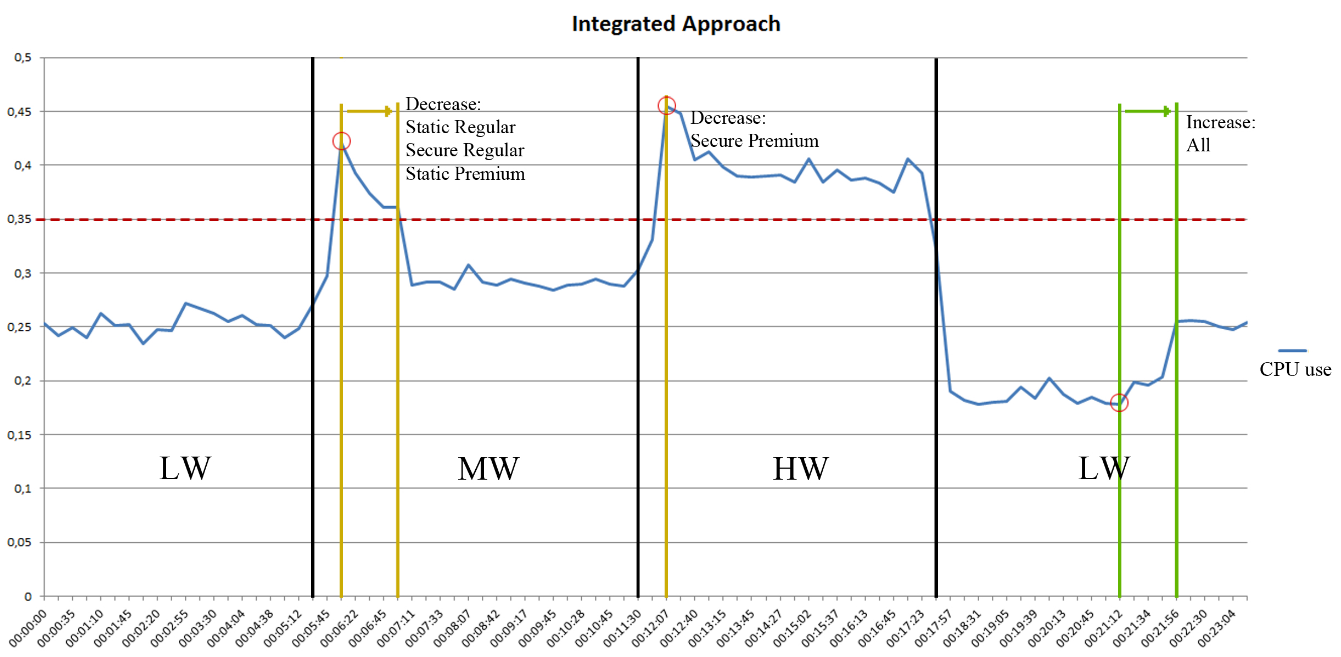

Figure 5.5: Results of the CPU Overload scenario on the Integrated Approach.

that can reduce the CPU use. As a result, during the High workload there is nothing more that can be done to solve the problem that has not been done using this Strategy. The GRA approach is not necessary in this scenario.

This test is interesting as it shows that the Regular services were not considered enough to fix the problem, and one of the Premium services was adapted as well. The reason for this is because of the stabilization period defined for each tactic. In this case it was not long enough for the effects of the two previous executed tactics to be observable, so the system executed an additional tactic, resulting in an excessive adaptation. However if the stabilization period is too long the strategy takes much longer to correct the problem.

Note that the same thing could have happened if the GRA approach was used during this test. The reason for that is that the workloads do not always result in the same exact behaviour on the server. The values measured over time during the same workload do not result in a straight line. In this case, the Medium workload resulted in a slightly higher CPU use than the one observed in previous tests, this means that adapting the Regular services might not have been considered enough by the GRA approach.

## 5.3.2 MRT Overload Scenario Results

Figure 5.6 displays the results of the MRT Overload scenario with the adaptation system using only the GRA approach.

The Light workload (LW) has the same results as in the previous scenario. The workload does not cause any problem on the system so no adaptation is required.

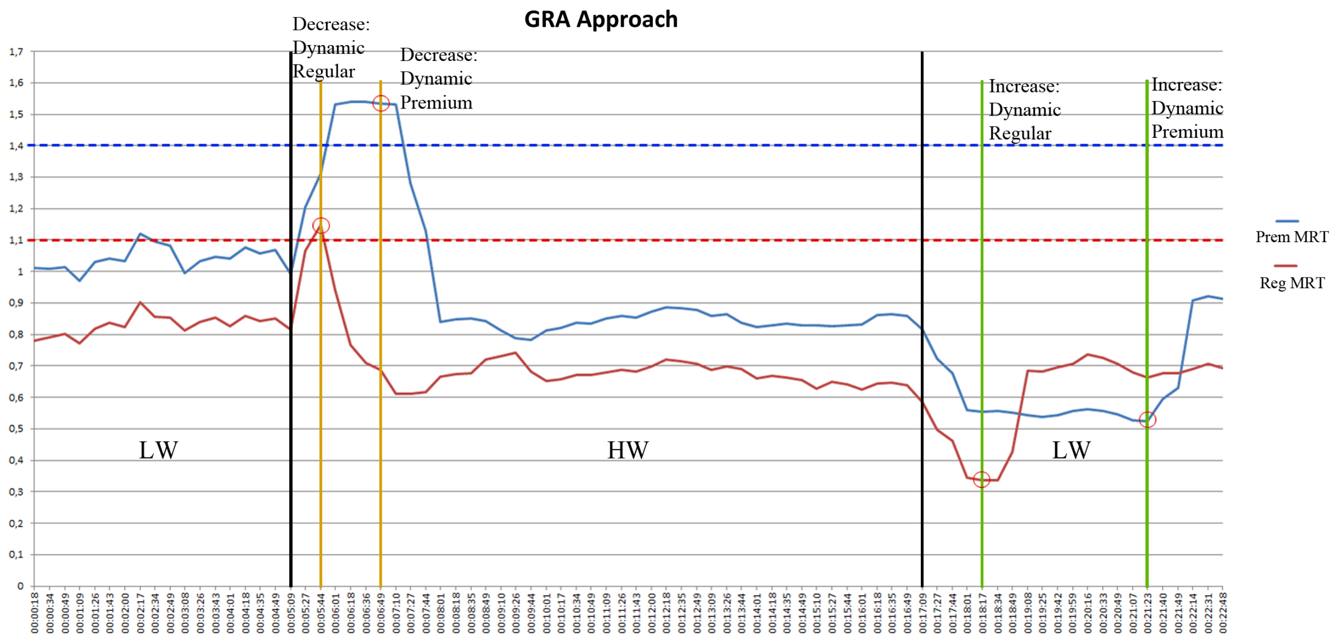

Figure 5.6: Results of the MRT Overload scenario on the GRA Approach.

The High workload (HW) however causes both Regular and Premium medium response time to go over their respective thresholds. The red dashed line is the threshold for the Regular services, the blue dashed line is the threshold for the Premium services. The system detects that the MRT of the Regular services goes above the threshold before the MRT of the Premium services. As a result, it adapts the Regular service before the Premium service.

When the Light workload is repeated the periodic event that refers to the maximize_qos_regular goal is triggered first which results in the Regular service being recuperated first.

Figure 5.7 displays the results of the MRT Overload scenario with the adaptation system using only the SDA approach. Because the Strategies specification does not include any Strategy designed to solve this problem, the adaptation system is unable to correct the problem.

Figure 5.8 displays the results of the MRT Overload scenario with the adaptation system using the Integrated approach. The behaviour observed in this test is very similar to the one observed when the system was using only the GRA approach. This happens because the Integrated approach is using the GRA approach to handle the problem.

One observable difference is that when the Light workload is executed again in the end, the periodic event that refers to the quality of service of the Premium services is triggered first.

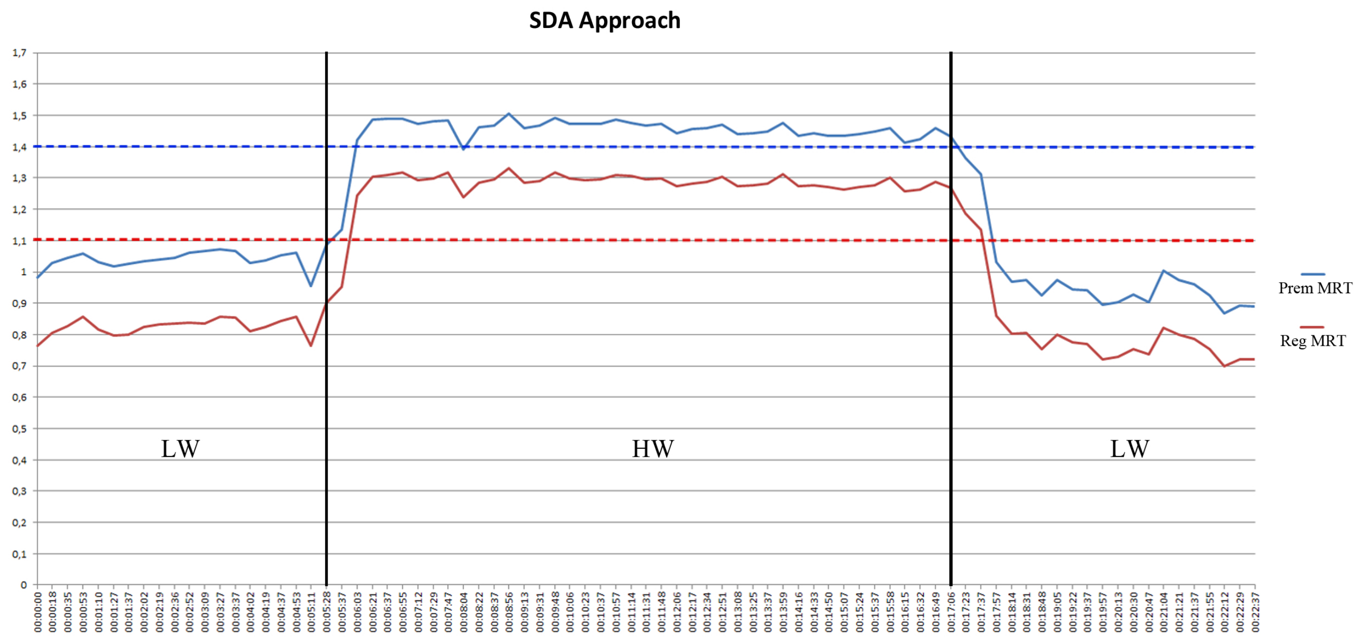

Figure 5.7: Results of the MRT Overload scenario on the SDA Approach.

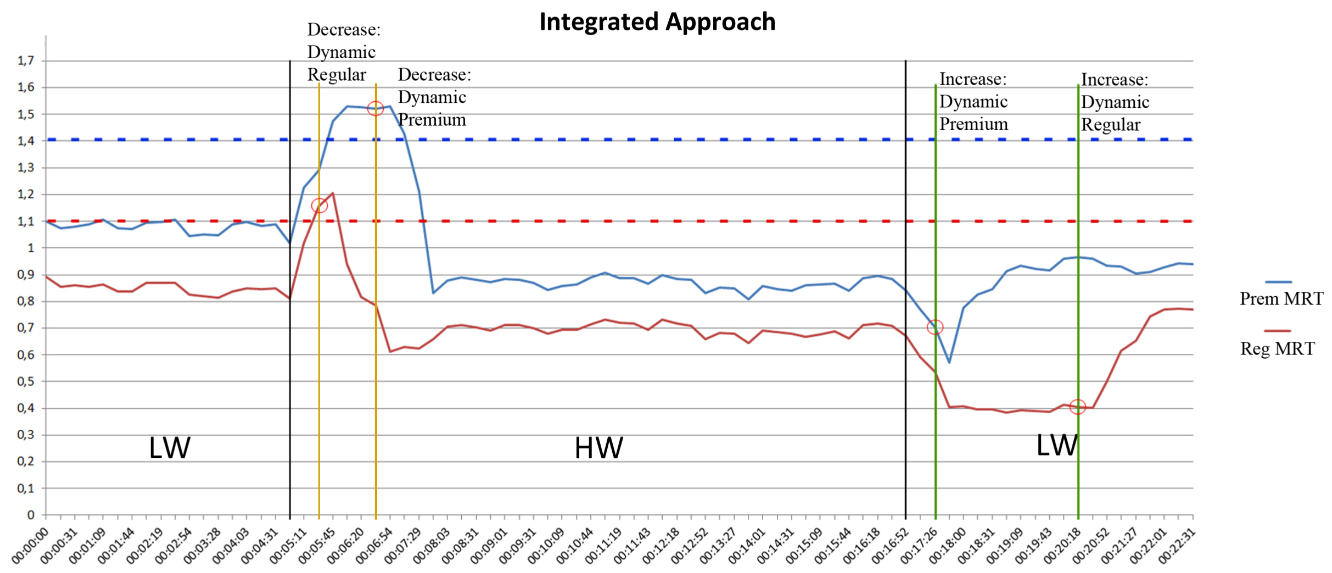

Figure 5.8: Results of the MRT Overload scenario on the Integrated Approach.

## 5.4 Alternative scenarios

In order to turn the evaluation more complete, we also considered some conceptual scenarios or changes to the scenarios presented in the Case Study and elaborate on the expected behaviour.

**Dynamic services impact CPU** During the CPU overload scenario the Integrated approach did not take advantage of the GRA approach. However, if the Dynamic service type adaptations had been defined with an impact on the CPU use the GRA approach would have found an additional adaptation plan that could further reduce the CPU use. As explained before, adaptations on the Dynamic service types have a small impact on the CPU use because most of the processing of these services is done on the Data server. However, the impact is not null, and using these adaptations might have been enough to bring the CPU use closer to the limit_cpu goal threshold.

**Incomplete Strategy** Consider the possibility that the FixCPU strategy was not implemented in a way that would allow the strategy execution system to get to the last edge when all the previous edges were not applicable. This means the SDA approach would not have been able to decrease the Secure Premium service when the workload changed to High. However because this adaptation is still applicable, the GRA approach would have been able to use it.

**Extended MRT overload** Over time the human operator will design a Strategy to solve the problem caused by the MRT Overload scenario. He could design it to mimic the behaviour observed in the results of the GRA approach. This means the problem is no longer unexpected and the Integrated approach will use the strategy the next time it happens. However, consider a new scenario where the load of the Static and Secure services would also be increased just enough to ensure the CPU use did not go over the threshold but the MRT does even when the Dynamic services are adapted. The system will adapt the Dynamic services using the strategy but that is still not enough. The Integrated approach would then use its knowledge that adapting the Static and Secure services can reduce the MRT and use the GRA approach to find a plan that involves adapting these services despite the fact that the CPU is not overloaded.

## 5.5 Results analysis

The knowledge of the functionality of the GRA, SDA and integrated approach and the analysis of the results of the performed experiments lead us to the following conclusions.

- The effectiveness of the SDA approach is similar to that of GRA approach on anticipated problems. While one depends on the “correctness” and effectiveness of the strategies that were provided to the system, the other depends on the accuracy of the estimates of impacts of adaptations. As the target system becomes more complex and the number of services and available adaptations increases, the performance of the GRA approach decreases — it becomes increasingly slower and heavier. The cost of using the SDA approach does not increase as fast when more strategies are included. However, as the number of services and KPIs increase, the knowledge of system operators becomes inevitably more limited and the complexity of the task of writing correct and effective strategies becomes extremely high. SDA approach is definitely appropriate for dealing with well known problems for which good strategies exist.

- The GRA approach is better than SDA in dealing with problems that do not occur very often and, hence, for which human operators do not have a repair strategy (situations which they would typically try to solve with a trial and error approach). However, for this to happen, GRA needs detailed and accurate information from component developers about the available adaptations and their impacts. GRA is also more flexible in what concerns the management of the importance of different system goals. GRA is better in situations in which the importance given to different goals may need to change. In SDA, as this importance is diluted on the strategy structure and its application conditions, this might require to rewrite the whole set of strategies.
- The Integration of both approaches can successfully use one approach or the other to solve both anticipated and non anticipated problems. This offers the human operators a platform to create and improve strategies to solve known problems over time while ensuring that the other problems (whose solution requires to have full detailed knowledge of the different components and their configuration options) can be handled as well.
- In the three approaches, the reaction time (the time between the workload change and the problem detection) depends on the checking period of the ExactEventLauncher and PeriodicEventLauncher. These checking periods can be configured to allow the system to react faster or slower. Faster reaction times result in executing the analysis process more often which increases the impact of the adaptation control layer on the target system.

Notice also that there is a delay between the time in which the problem becomes visible in the State and when the adaptations take place. This delay is caused by multiple factors. The first factor is the time between when the StateManager updated the state and when the EventLaunchers checked whether that state violates any goals. The second factor is the time between when the event was launched and when the adaptations were executed. Finally, the third factor is the time between the execution of the adaptations and when their effects become observable.

The time it takes for each approach to begin executing adaptations after an event has been launched is hard to measure without greatly increasing the number of available adaptations and strategies. However, we can analyse this conceptually:

- The SDA approach has to go through the applicability conditions of all its strategies to find the applicable ones and then pick one. This process should be fast but the actual execution of a strategy can become slow depending on how many tactics it has to execute and their individual stabilization periods.

- The GRA approach has to calculate the expected impact of all the adaptation plans of the adaptation rule that matches the event. Then it has to apply the goal filters to those adaptation plans. This process should be slower than the process of the SDA approach. However, because the system calculates the expected impact of the adaptation plan it can execute all necessary adaptations at the same time. If the impact functions are accurate enough a single adaptation plan will be enough and the execution of this plan is faster than the execution of the Strategy.
- The Integrated approach has to do the same as the SDA approach. On top of that it executes all applicable strategies in sequence until the problem is solved. This means that when a Strategy fails but another applicable strategies exist the system does not need to wait for the next event to be launched. If there are no applicable strategies or the strategies failed to solve the issue the approach does the same as the GRA approach.

# Chapter 6

# Conclusions

This dissertation presents a way to integrate two different and complementary approaches to self-adaptation—GRA and SDA. GRA is a goal-oriented approach that relies on the machines superior computation capabilities to produce an adaptation plan in real time when a problem is detected. SDA is a strategy-oriented approach that uses human designed strategies to solve problems when they arise. The integration of these approaches has the potential to harness the advantages of both while mitigating their disadvantages.

The approach to the integration presented in this paper gives the human operator the possibility to design his own solutions to known problems while still allowing the system to calculate an adaptation plan by itself to solve unknown problems. The SDA approach is a simplified version of the approach described in the Stitch paper [7]. Using this approach, the human operator is capable of designing Strategies without having to know all available adaptations and their impacts on the system.

The fact that the integrated approach chooses to use available strategies first and automatically generated adaptation plans only when strategies are not enough gives the human operator control over the adaptation behaviour. The approach is also designed to allow the strategies to be edited or added at any time.

The result is an adaptation system that is capable of solving problems by itself but will favour human designed solutions over real time automatically calculated ones. The advantages identified in section 2.5 for both the goal-oriented and strategy-oriented approaches are included in their integration. However, the disadvantages have been mitigated.

## 6.1 Future Work

There is still some work that can be done on the implementation of the approach. Some changes would result in trade-offs while others would simply improve the existing prototype:

- In the current implementation the Tactic is a overly simplified adaptation. It could be useful to allow the Tactics to be implemented with impact functions like in the case of Adaptations. This would allow us to use much more intelligent Strategy selection algorithms, however it would also mean the Strategies would have to be Trees so that their expected impact could be calculated. This change would complicate the definition of the Strategy.

- The logic that accomplishes the execution of an Action should be encapsulated on the Action itself. This would allow the human operators to load the system with effectors in the form of Actions.
- The way the sensor reader returns the information and the way the state manager stores it could be greatly improved to allow easier configuration for a system administrator.
- A way to load the system with goals, available adaptations, tactics and strategies could be developed. This way the adaptation logic would be completely separated from the target system specification and expected behaviour given by the system administrator.

# Bibliography

[1] Ibm: An architectural blueprint for autonomic computing.

[2] ADAAS. See `adaas.dei.uc.pt/adaas`.

[3] K.J. Astrom. Adaptive feedback control. *Proceedings of the IEEE*, 75(2):185–217, Feb. 1987.

[4] Raphael M. Bahati, Michael A. Bauer, and Elvis M. Vieira. Policy-driven autonomic management of multi-component systems. In *CASCON '07*, pages 137–151, NY, USA, 2007. ACM.

[5] Arosha K. Bandara, Emil C. Lupu, Jonathan Moffett, and Alessandra Russo. A goal-based approach to policy refinement. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:229, 2004.

[6] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.

[7] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Stitch: A language for architecture-based self-adaption.

[8] Shang-Wen Cheng, Vahe V. Poladian, David Garlan, and Bradley Schmerl. Software engineering for self-adaptive systems. chapter Improving Architecture-Based Self-Adaptation through Resource Prediction, pages 71–88. Springer-Verlag, Berlin, Heidelberg, 2009.

[9] Y. Diao, J. L. Hellerstein, S. Parekh, and J. P. Bigus. Managing web server performance with autotune agents. *IBM Syst. J.*, 42(1):136–149, 2003.

[10] Ahmed M. Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *SIGSOFT FSE*, pages 7–16, 2010.

[11] Betty H. C. Cheng et al. Software engineering for self-adaptive systems: A research roadmap. In Cheng et al. [6], pages 1–26.

[12] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *IEEE Softw.*, 23:62–70, March 2006.

[13] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, October 2004.

[14] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In Johnson [17], page 7.

[15] J.C. Georgas. Supporting architecture- and policy-based self-adaptive software systems, 2008.

[16] Lars Grunske, Ralf Reussner, and Frantisek Plasil, editors. *Component-Based Software Engineering, 13th International Symposium, CBSE 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings*, volume 6092 of *Lecture Notes in Computer Science*. Springer, 2010.

[17] J. Howard Johnson, editor. *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*. IBM, 1997.

[18] J.O. Kephart and W.E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3 – 12, june 2004.

[19] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in a reflective component model. In Grunske et al. [16], pages 74–92.

[20] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37:56–64, July 2004.

[21] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44 –51, oct. 2009.

[22] Hausi A. Müller, Holger M. Kienle, and Ulrike Stege. Software engineering. chapter Autonomic Computing Now You See It, Now You Don't, pages 32–54. Springer-Verlag, Berlin, Heidelberg, 2009.

[23] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54 –62, may/jun 1999.

[24] Liliana Rosa, Luís Rodrigues, Antónia Lopes, Matti Hiltunen, and Richard Schlichting. From local impact functions to global adaptation of service compositions. RT 35/2011, INESC ID Lisboa, 2009.

[25] Liliana Rosa, Luís Rodrigues, Antónia Lopes, Matti A. Hiltunen, and Richard D. Schlichting. From local impact functions to global adaptation of service compositions. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 5873 of *Lecture Notes in Computer Science*, pages 593–608. Springer, 2009.

[26] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[27] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of caching and replication strategies for web applications. *Internet Computing, IEEE*, 11(1):60–66, Jan.-Feb. 2007.

[28] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-managing systems*, pages 1–8, NY, USA, 2008. ACM.

[29] Gerald Tesauro and Jeffrey O. Kephart. Utility functions in autonomic systems. In *Proceedings of the First International Conference on Autonomic Computing*, pages 70–77, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Jie Yang, Gang Huang, Wenhui Zhu, Xiaofeng Cui, and Hong Mei. Quality attribute tradeoff through adaptive architectures at runtime. *J. Syst. Softw.*, 82:319–332, February 2009.

[31] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS '02*, page 301, Washington, DC, USA, 2002. IEEE Computer Society.

[32] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 301–, Washington, DC, USA, 2002. IEEE Computer Society.