Evolving Software Systems for Self-Adaptation

by

Mehdi Amoui Kalareh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There is a strong synergy between the concepts of evolution and adaptation in software engineering: software adaptation refers to both the current software being adapted and to the evolution process that leads to the new adapted software. Evolution changes for the purpose of adaptation are usually made at development or compile time, and are meant to handle predictable situations in the form of software change requests. On the other hand, software may also change and adapt itself based on the changes in its environment. Such adaptive changes are usually dynamic, and are suitable for dealing with unpredictable or temporary changes in the software's operating environment.

A promising solution for software adaptation is to develop self-adaptive software systems that can manage changes dynamically at runtime in a rapid and reliable way. One of the main advantages of self-adaptive software is its ability to manage the complexity that stems from highly dynamic and nondeterministic operating environments. If a self-adaptive software system has been engineered and used properly, it can greatly improve the cost-effectiveness of software change through its lifespan. However, in practice, many of the existing approaches towards self-adaptive software are rather expensive and may increase the overall system complexity, as well as subsequent future maintenance costs. This means that in many cases, self-adaptive software is not a good solution, because its development and maintenance costs are not paid off. The situation is even worse in the case of making current (legacy) systems adaptive.

There are several factors that have an impact on the cost-effectiveness and usability of self-adaptive software; however the main objective of this thesis is to make a software system adaptive in a cost-effective way, while keeping the target adaptive software generic, usable, and evolvable, so as to support future changes. In order to effectively engineer and use self-adaptive software systems, in this thesis we propose a new conceptual model for identifying and specifying problem spaces in the context of self-adaptive software systems. Based on the foundations of this conceptual model, we propose a model-centric approach for engineering self-adaptive software by designing a generic adaptation framework and a supporting evolution process. This approach is particularly tailored to facilitate and simplify the process of evolving and adapting current (legacy) software towards runtime adaptivity. The conducted case studies reveal the applicability and effectiveness of this approach in bringing self-adaptive behaviour into non-adaptive applications that essentially demand adaptive behaviour to sustain.

Acknowledgements

First and foremost, I want to thank my supervisor Professor Ladan Tahvildari. I appreciate all her contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm she has for research was contagious and motivational for me, even during tough times in the Ph.D. pursuit.

I wish to thank the members of my dissertation committee: Professor Marin Litoiu, my external examiner, for having accepted to take the time out of his busy schedule to read my thesis and provide me with his insightful suggestions; Professor Patrick Lam for his invaluable comments and feedback on the thesis and traveling from Berkeley to attend my defense; Professor Kostas Kontogiannis for his inspiring remarks, and also traveling all the way from Europe to participate my defense session; Professor Michael Godfrey for his insightful feedback during different stages of my Ph.D.; and Professor Daniel Berry for serving as a delegate in my defence session.

I would like to express my gratitude to Professor Jürgen Ebert for supporting the GRAF project with his insightful comments and feedback; and most importantly giving me the opportunity to do a close collaboration with the Institute for Software Technology (IST) at the University of Koblenz-Landau, and his kindest support and hospitality during my stay at Koblenz as a visiting scholar.

I feel very lucky to have been introduced to and have a chance to work with Mahdi Derakhshanmanesh. Mahdi is a great software engineer and an awesome friend. It was my pleasure working with him closely, and I am thankful for all the time and effort he has put on the GRAF project.

I would like to thank all the member of Software Technologies and Applied Research (STAR) Group for their moral support and valuable feedbacks. In particular, I would like to thank: Dr. Mazeiar Salehei for his brotherly support and all the brainstorming sessions, discussions, and memorable times we had together in the past few years. Siavash Mirarab for being a remarkable friend and supportive, specially in the early days of my graduate studies at Waterloo. Sen Li, for his friendship and thoughtful support during the long hours we had in the lab. And Greg O'Grady for his effort on the case studies presented on this thesis.

I owe an immense debt of gratitude to my family for all their love and encouragement. To my parents who raised me with love and supported me in all my pursuits. And to my caring, supportive, encouraging, and patient friend, Niousha, whose faithful support during the final stages of this Ph.D. is so appreciated. Thank you.

Dedication

To my beloved parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"*It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.*"
–Charles Darwin

Many engineering solutions are inspired by nature. One of these inspirations is *adaptation*; that is, the way living organisms change their structure and/or behaviour in order to retain or achieve a better compatibility with their environment [53].

Adaptation can occur either instantly or over time. Organisms may adapt themselves to changes in their environment as these changes take place. This type of alteration is limited to an organism's *adaptedness*; that is, the degree to which an organism is able to live and reproduce in its environment [53]. Thermoregulation and learning are examples of these kinds of adaptation in living organisms. On the other hand, *evolution* is a change process that happens continuously over time throughout generations. The result of evolution is a new individual that carries most of the properties of its base, to the extent that it remains in the same class as its base. Evolution happens gradually, and the resulting newer generations are expected to be better individuals in terms of functionality and quality. Therefore, adaptation refers to both the current organism being adapted and to the evolutionary process that leads to the adapted generations. Evolution can result in adaptation, but not all evolution changes adapt an organism to its surroundings, even though evolution may increase adaptedness. This phenomenon is known as an organism's *adaptive trait*; i.e., an aspect of the developmental pattern of the organism that enables or enhances the probability of that organism's survival and reproduction [52].

Software engineering as an engineering discipline is the profession of applying scientific knowledge and utilizing natural laws, in order to design and implement software systems

and processes that realize a desired objective and meet specified criteria [168]. Software engineering has borrowed the concepts of adaptation and evolution from nature. *Software adaptation* is a process in which a software system changes its structure and/or behaviour in order to retain or achieve better compatibility with its environment. *Software evolution* is the phenomenon of a software system changing over time, similar to the evolution of living organisms. However, evolution in the context of software has a slightly different meaning from its original definition in biology. In software engineering, evolution is usually accompanied by maintenance, in which the arbitrary nature of evolutionary changes are replaced with supervised and goal-oriented ones (e.g., changes for the purpose of adaptation, known as adaptive maintenance).

Evolution changes for the purpose of adaptation are usually supervised and occur at development or compile time due to requirement changes or discovered bugs. These changes are usually extensive and happen in almost every instance of a system over time, and are meant to handle predictable situations in the form of software change requests. Evolutionary changes are generally stable, but in most cases they are costly and require a significant amount of time and effort to apply. These changes may also serve as an *adaptive trait* for increasing adaptedness in software systems. *Adaptedness* in software systems describes the degree to which software is able to operate and evolve in dynamic and new operating environments. This is a favorable property of software systems that operate in dynamic and nondeterministic environments. On the other hand, software may also change and adapt itself according to the changes in its environment. Such adaptive changes are usually dynamic: The software changes at runtime based on instant decisions. Therefore, adaptive changes are suitable for dealing with unpredictable or temporary changes in the software's operating environment.

Despite the above differences between evolution and adaptation, there is a strong synergy between both concepts in software engineering. Similar to living organisms, software adaptation refers to both the current software being adapted and to the evolution process that leads to new adapted software, due to an alteration in the requirements or the operating environment. This thesis elaborates this synergy by proposing a novel engineering approach realizing the required adaptations in software systems.

1.1 Motivation

As software ages, it should change with time, or else it will not survive [112]. Software changes can be grouped as follows, based on the purpose that they serve: They may be

corrective for fixing bugs, *adaptive* for adapting the software to new environments, *perfective* for updating the software according to requirements changes, and *preventive* for making the software more maintainable [88]. However, software maintenance and evolution are inevitably costly and time-consuming activities for almost any software-intensive system. From an economic point of view, system adaptation, on average, soaks up most of the post-delivery non-corrective software costs ($\sim$ 80% of total post delivery costs are non-corrective) in the life of a software system [167]. Moreover, a key characteristic of the maintenance/evolution of large software systems is that change becomes increasingly difficult and expensive over time [13].

Since the era of software engineering started, many technologies and development paradigms have been introduced to scope, estimate, control, and improve the predictability and efficiency of software change [30, 37, 83]. These approaches aim at reducing the cost of change by either (i) managing the total number of the required changes, (ii) making software change easier (e.g., creating more flexible and variable software), (iii) reducing software down-times, and (iv) predicting the change.

A promising solution to reduce the cost of software change is to develop *self-adaptive software (SAS)* systems that are able to manage changes dynamically at runtime in a rapid and reliable way. One of the main advantages of SAS is its ability to manage the complexity that stems from highly dynamic and nondeterministic operating environments. Frequently changing user needs or high variability in the amount of available resources are two possible scenarios in which SAS can reconfigure and continuously optimize itself at runtime. An adaptive behaviour can replace a normal behaviour in an autonomous way.

In general, a SAS follows a reference architecture that comprises two main subsystems [98]: (i) the adaptation manager (autonomic manager), and (ii) the adaptable software (managed element). The adaptation manager acts as an external controller that observes the adaptable software for changes in its operating states and selects appropriate adaptive behaviour accordingly. These two subsystems are connected through sensor and effector interfaces, as shown in Figure 1.1.

.

1.2 Problem Description

A software system has a set of properties, including its internal properties, its input and output, and the relationships between the inputs and outputs. These properties are captured in the form of specifications, as the desired properties of the software system. How-

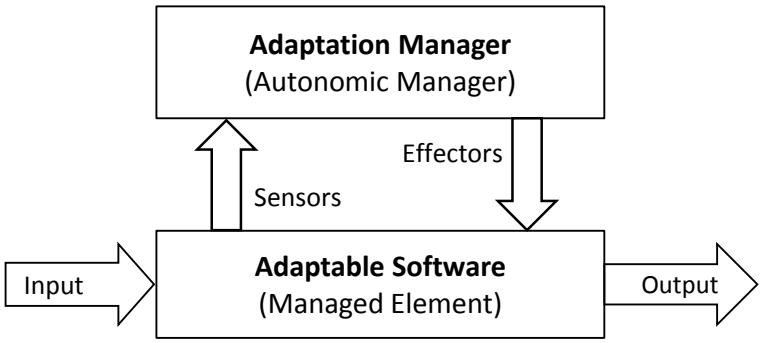

Figure 1.1: Reference Architecture of a Self-adaptive Software (SAS) [98]

ever, in reality the operational software may not fully match its specification. Moreover, as a software system ages, its environment and user needs will change, which will result in an increase in the deviation of the software's properties from its desired properties. This phenomenon is captured by the first law of software evolution by Lehman [112]: "Software change is inevitable and systems must be continuously adapted."

In classic software engineering, all required post-delivery changes are handled through evolution changes that result in a new software system. In the case of self-adaptive software, some of the required changes can be handled through adaptation changes, which can change the properties of the software dynamically without generating a new software system. The adaptation changes impose less cost compared to their equivalent evolution changes, and are usually invertible.

In case the original software system is not adaptive, we can apply a set of evolution changes, which serve as adaptive traits that increase the adaptedness of the software system. As the environment and/or user needs change, the desired properties of the software including its adaptedness may change overtime. Hence, evolution changes that introduce and/or improve adaptedness are not one-time changes: We need continuous evolution to maintain the desired adaptedness in software.

Each change has a cost. The effectiveness of a change is determined by the degree to which the change meets its purpose of achieving the desired properties of the software throughout the software's lifespan. The objective of systems engineering is to ensure that a system is designed, built, and operated so that it accomplishes its purpose in a cost-effective manner. Cost-effectiveness considers both the cost and the effectiveness of a system in the context of the system objectives.

If a SAS system has been engineered and used properly, it can greatly improve the

cost-effectiveness of software change through its lifespan. However, in practice, many of the existing approaches towards SAS are rather expensive and may increase overall system complexity, as well as subsequent future maintenance costs. This means that in many cases SAS is not a good solution, because its development and maintenance costs are not being paid off. The situation is even worse in the case of making current (legacy) systems adaptive. Hence, the most important question to be answered is:

"How can we effectively engineer and employ self-adaptive software?"

There are several factors that have an impact on the cost-effectiveness and usability of self-adaptive software; however, the main objective of this thesis is to make a software system adaptive in a cost-effective way, while keeping the target adaptive software generic, usable, and evolvable, so as to support future changes.

1.3 Research Challenges

There are a number of challenges associated with the objectives of this thesis, which are clustered into the following categories:

Managing the Complexity of Self-Adaptive Software

SAS systems are complex systems to design, build, and maintain; including self-adaptive and autonomic behaviour into an existing system comes at the price of additional complexity. For many real-world scenarios, this undesired *accidental complexity* [23] inhibits the pursuit of runtime adaptivity. The benefit of reducing essential complexity must be less than the harm of any augmented accidental complexity, which is a problem that many advanced software-engineering solutions fail to address.

To tackle the complexity issue in the context of SAS, the construction of precise and accurate models that address the demands of self-adaptive systems is a necessary but hard task, given highly complex and dynamic nature of SAS systems and their operating environment [6, 38]. Such models must cover the complete engineering cycle. There is a need to derive domain-specific models of software, ranging from problem-space models to support adaptation-requirements specification, to solution-space models that are to be generated, manipulated, and managed at runtime. These models must embody the adaptive responsibilities of SAS and serve as a solid basis for further adaptation. However, current

metamodels are not well suited for engineering SAS, and fail to unify adaptation aspects with available knowledge about the operating environment and constituent application entities.

Preparing Self-Adaptive Software

Another downside of moving towards SAS is that it is costly and hardly possible to make a software system adaptive without changing the original software. We are interested in making current systems adaptive and integrating them with high-end technology trends. However, this is a more demanding process than developing adaptive software from scratch, because most of the existing and legacy systems are developed based on older technologies and tools.

Here, the question is how to reengineer software towards adaptability concerns. These concerns include: (i) having a mechanism to sense any changes in inter or intra software states, (ii) deciding and selecting the best action for changing a software system in order to adapt to environment changes, and (iii) having proper effectors to change the software dynamically at run-time. Nevertheless, there is no generic solution to the mentioned concerns, and hence no clear and solid answer to the following questions: (i) *What* are the sensors and effectors? (ii) *Where* should the sensors and effectors be placed in software? (iii) *When* should the sensors be active? (iv) *How* should required data be sensed, and desired actions be effected?

Planning for Software Changes

There is no unique way to change software for the purpose of adaptation. Valid solutions can be a combination of adaptation and evolution changes, and there are tradeoffs between different solutions. In general, the cost of evolution is higher than adaptation, due to the fact that evolution results in a new system/sub system, which increases development and maintenance costs.

Choosing the best approach requires knowledge about changes that might take place in future. Thus, it is important to predict future changes and estimate the effort required to perform these changes. Another important concern is how a software system will perform and react in a new situation or environment. For example, if an upcoming change request (that is likely to happen in the future) is known at the present time, then the appropriate behaviour in that situation shall be considered a functional requirement. However, in many scenarios, this information is not available, or its occurrence is very unlikely, implying that

the cost and effort of handling the situation does not payoff. Consequently, for any given change request, we have to consider that this is not the last change request, and that similar change requests may be required in future. Yet, we still need to investigate the possibility and effectiveness of software-change prediction models, and how we can benefit from them to plan for evolution and adaptation changes.

1.4 The Approach

The mentioned research objectives and problems associated with engineering and using SAS systems call for a well-defined approach for improving the usability and effectiveness of SAS. This demand influenced the original idea of this thesis, which was to design and develop a generic adaptation framework and a supporting evolution process, so as to manage the complexity of both the final SAS system and the engineering process for constructing it.

Complexity in SAS stems from various sources (e.g., complexity of software, resources, communication channels, adaptation logic, etc.). In this thesis, we use two approaches to manage the complexity of SAS: (i) using models to support various stages of the SAS life-cycle, ranging from models for specifying adaptation requirements to models that are to be generated, manipulated, and managed at runtime; and (ii) balancing the complexity among its sources by finding pareto solutions that can minimize the total system complexity.

For this purpose, we initially separate *adaptation* and *adaptability* concerns and identify key conceptual entities in both of the adaptation and adaptability problem spaces. Then we link these entities together as a unified conceptual model for specifying SAS systems. This reference model is a composition of domain-specific metamodels that can characterize and enclose particular aspects of SAS to provide a better understanding of the SAS and its domain.

Part of the proposed conceptual model is the metamodel for software adaptability. This metamodel helps developers comprehend and evolve software for adaptability, and assists in the construction of transformations aimed at adding adaptive behaviours and improving software adaptability. The metamodel provides new modeling elements for capturing adaptability properties of software in terms of a set of primitive sensing and effecting styles.

The second part of our approach is to use models of software at runtime for the purpose of adaptation. In this approach to runtime adaptivity, the software to be controlled at runtime is the *adaptable software*. It is connected to a meta-layer, which is the actual *runtime model*. Moreover, an *adaptation manager* controls the adaptable software by manipulating its runtime model instead of directly operating on the adaptable software.

The structure and behaviour of software that is built around this *model-centric architecture* can be changed by modifying the models only. In addition, model transformations can support the implementation of adaptivity by modifying models at runtime.

In collaboration with the Institute for Software Technology at the University of Koblenz-Landau, Germany, we realize and implement the proposed model-centric adaptation approach as the *Graph-based Runtime Adaptation Framework* (GRAF), which utilizes TGraphs and its accompanying technologies as the enabling technology for modeling and manipulating a runtime model. GRAF explicitly separates the adaptable software from its runtime model, so as to isolate adaptivity concerns from the rest of the common business logic. In addition, the framework supports the separation of supervision and control from the application's core functionality. This is achieved by realizing the adaptation manager externally, instead of mixing it with the adaptable software itself.

The developed model-centric approach can be applied to creating an SAS by either (i) *developing* it from scratch, or (ii) *modernizing* existing software as a special case of developing an SAS anew. In this regard, the design of GRAF fulfills the requirements for an evolution towards self-adaptive software. However, a modernization requires additional steps to complete a reengineering process for making an original application adaptable. Therefore, as a part of this thesis, we propose a modernization process that is specially tailored to (re)engineer software systems towards model-centric self-adaptivity. The proposed approach uses GRAF in conjunction with other techniques and tools.

The modernization process starts by analyzing adaptation requirements and the current system and its application domain in order to specify the required adaptability and adaptation. After the analysis step, we plan for changing the original software and specify the required evolution changes in the form of program transformations that preserve the original behaviour of the software when there is no need for adaptation. The process also includes the preparation of the adaptation manager (i.e., customized GRAF), and its integration with the evolved software, such that the software can operate independently from its adaptation manager to provide its original behaviour.

The approach is supported by conducting a number of case studies that serve as a proof of concept, show by example, how to achieve runtime adaptivity with GRAF, and sketch the framework's capabilities for facilitating the modernization of real-world applications towards a self-adaptive software system in a cost-effective way.

1.5 Thesis Contributions

Regarding the objectives of this research work and our approach, the following contributions are presented in this thesis:

- *A Unified (Reference) Conceptual Model for Controller-based SAS:* This is a model of the SAS problem space that identifies key concepts in this problem space and elaborates their roles in SAS specifications. The reference model particularly considers shared phenomena between the adaptable software, its adaptation manager, and its environment. Since specifications and architecture impact each other, this contribution also includes a discussion of solution-space considerations and how the reference model may facilitate design.

- *A Metamodel For Software Adaptability:* This metamodel assists developers in comprehending and evolving software for adaptability, and assists the construction of transformations for including adaptive behaviours and improving software adaptability. This metamodel provides new modeling elements for capturing the adaptability properties of a software system in terms of a set of primitive sensing and effecting styles.

- *A Model-Centric Approach for Self-Adaptation:* This approach extends proven approaches from the domain of model-centric and model-driven development, and enables adaptivity by creating, managing, and interpreting runtime models of software. The approach is especially suited for the modernization of current (legacy) applications towards runtime adaptivity by reducing the changes necessary to make software adaptable. This approach is first published in [2], and realized as the *Graph-based Runtime Adaptation Framework (GRAF)*.

- *A Methodology to (Re)Engineer Software Towards Model-Centric Self-Adaptivity:* This is a modernization process for evolving software systems towards GRAF-based SAS. The proposed approach uses GRAF in conjunction with other techniques and tools to achieve this goal. In this methodology, adaptability specifications and adaptation specifications are co-evolved to reduce evolution and adaptation costs. Additionally, the cost of evolution changes are kept minimal by defining the required evolution changes in the form of behaviour-preserving transformations. The methodology also includes the steps to prepare and integrate GRAF with the evolved software.

- *Case Studies:* This thesis presents case study results that show how to achieve runtime adaptivity with GRAF, and sketch the framework's capabilities for facilitating the modernization of real-world applications towards self-adaptive software in a cost-effective

way. The case studies also provide some details of the GRAF implementation and evolution process, and examine their usability and performance in real-world applications.

This research has also led to several valuable by-products, which are not directly in the scope of the defined problem, but can be utilized and used as an extension of this thesis:

- *Adaptive Action Selection Technique Using Reinforcement Learning* [4]: This contribution proposes a planning process, and specifically, an action-selection technique based on *Reinforcement Learning* to select appropriate actions in highly dynamic environments.
- *Temporal Software-Change Prediction Using Neural Networks* [5]: This contribution includes a software-change prediction technique, and a supporting framework, for predicting the occurrence of several future software changes with reasonable performance. This approach provides a new viewpoint to managers and developers for planning maintenance activities more efficiently. The approach indicates *where* the changes are likely to happen, and then adds the time dimension to predict *when* it may occur.

1.6 Thesis Organization

This document is further organized as follows:

- Chapter 2 describes the related background knowledge for this thesis, and explores related work, with the aim of putting the thesis in context. The chapter covers three research fields that form the foundation of this thesis: *software maintenance and evolution*, *self-adaptive software*, and *engineering aspects of self-adaptive software*.
- Chapter 3 proposes a unified reference model for the SAS problem space by identifying key concepts in this problem space and elaborating their roles in SAS specifications. Since specifications and architecture impact each other, we also discuss solution-space considerations and how the reference model may facilitate design.
- Chapter 4 proposed the model-centric adaptation approach, and elaborates on the concept of model-centric architecture and models at runtime in SAS systems, with an emphasis on the design considerations and motivating factors for the approach. Additionally, the chapter gives an overview of the *Graph-based Runtime Adaptation Framework (GRAF)* and provides a brief summary of the framework's runtime behaviour and the technologies that are involved in the framework's implementation.

- Chapter 5 introduces the proposed modernization process model for evolving current (legacy) software towards model-centric self-adaptive software.
- Chapter 6 presents the case studies and obtained results.
- Chapter 7 presents the conclusions and discusses future research directions in the area of evolution and adaptation of model-centric self-adaptive software systems.
- Appendix A is a list of refactorings that are used by the transformations for making an original application adaptable.
- Appendix B presents the runtime model schema that is used in the current implementation of GRAF.

Chapter 2

Background Concepts and Related Work

"*Make everything as simple as possible, but not simpler.*"
–Albert Einstein

As highlighted in the introduction, there is a strong synergy between evolution and adaptation in software engineering. With the aim of putting this thesis in context, this chapter presents related concepts from software evolution, self-adaptive software, and engineering aspects of self-adaptive systems that shape the foundations of this thesis. The relationship between these concepts is depicted in Figure 2.1.

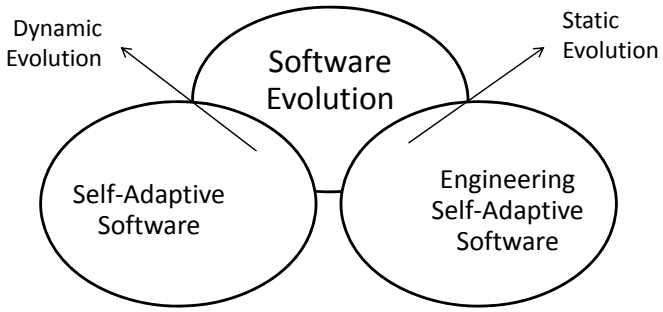

Figure 2.1: Thesis in the Context of Software Evolution

As shown in Figure 2.1, in this thesis we benefit from software evolution concepts and techniques in two different directions: (i) we use dynamic software evolution, model trans-

formation, and model generation as a part of our model-centric self-adaptation approach, and (ii) classic (static) software evolution to support our proposed modernization process for evolving software systems towards self-adaptation.

The rest of this chapter starts by giving an overview of fundamental concepts in software evolution and the process of changing software for various purposes. As software evolution is a broad discipline, this section continues with a focus on the concepts of software evolution that shapes the foundations of this thesis. The relationship between these concepts and their position in software change is depicted in Figure 2.2. Next, we present basic principles, properties, and background behind self-adaptive software, and then we elaborate engineering aspects of self-adaptive software. Finally, the closing section highlights the challenges and obstacles against the objective of this research to effectively engineer and use self-adaptive software.

2.1 Software Evolution

It is well accepted that the software development process is not a one time task resulting in perfect software. The first release of software may change during its life-cycle due to various reasons. Any changes and modifications on initial software are categorized under software maintenance and evolution, in order to adapt an application to ever-changing user requirements and operating environment. The necessity and characteristics of software change are defined as Lehman's laws of software evolution [112].

Many researchers and practitioners consider software evolution as a preferable substitute for software maintenance[1] [13, 37, 113], as an activity to perform post-delivery software changes [115], while some others refer to software maintenance as the process of fixing bugs or performing minor enhancements, and software evolution as coarser grained structural forms of change that makes the software systems qualitatively easier to maintain [188, 196]. In latter perspective, maintenance preserves the structure of the system with few economic and strategic benefits and evolution allows the system to comply with broad new requirements and gain whole new capabilities.

From another point of view, software evolution goes beyond software maintenance. Evolution techniques can be used to perform maintenance tasks such as bug fixing or any other adaptation within the software's original feature set. Moreover, evolution can

[1]Software maintenance as defined in IEEE Standard 14764 [88] is: "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

also be applied for changing systems in a more substantial way, for instance, to add new functionalities. With the view of having evolution as a top level activity of any post-release software change, software evolution can be divided into three categories: maintenance, modernization, and replacement [167]. Based on the above definition, we can conclude that software evolves as we maintain it.

Buckley *et al.* [30] classify various dimensions of software change by characterizing the mechanisms of change and the factors that influence these mechanisms. These dimensions subdivide software change into four logical themes: temporal properties (*when*), object of change (*where*), system properties (*what*) and change support (*how*). The captured dimensions, as a taxonomy of software change, are listed in Table 2.1, which can be used to compare formalisms or processes for software evolution. Among the dimensions listed in Table 2.1, in this thesis we are specifically interested in the *Time of Change* dimention, in which three categories of change become apparent:

- *Static:* The change concerns the source code of the system. Consequently, the software needs to be recompiled for the changes to become available.
- *Load-time:* The change occurs while software elements are loaded into an executable system.
- *Dynamic:* The change occurs during execution of the software.

Static and dynamic adaptations are mapped respectively to compile-time evolution and load-time/run-time evolution. That is why dynamic adaptation is sometimes called *dynamic evolution*.

The rest of this section gives an overview on software modernization and other related concepts that are utilized in this thesis. The relationship between these concepts and their position in the context of software evolution is depicted in Figure 2.2.

2.1.1 Software Modernization

Modernization is a type of evolution that involves more extensive changes than maintenance, while conserving a significant portion of the existing system [167]. Modernization changes can be categorized as either adaptive or perfective maintenance changes according to the Lientz and Swanson classification [115]. These changes often include restructuring the system, enhancing and adding new functionality, or modifying software attributes. The main objective of modernization is adapting the software to support new technologies and

Table 2.1: Dimensions Characterizing or Influencing Software Change Mechanisms [30]

Theme	Dimension	Description
When	Time of Change	addresses the when question.
	Change History	refers to the history of all (sequential or parallel) changes that have been made to the software.
	Change Frequency	influences the change support mechanisms. Changes to a system may be performed continuously, periodically, or at arbitrary intervals.
	Anticipation	refers to the time when the requirements for a software change are foreseen.
Where	Artifact	software artifacts can be subject to change during evolution. These can range from requirements through architecture and design, to source code, documentation, configurations and test suites.
	Granularity	refers to the scale of the artifacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity.
	Impact	the impact of a change, ranging from local to system-wide changes.
	Change Propagation	the process of following-up changes in other encapsulated entities (procedures, modules, classes, packages, etc.).
What	Availability	indicates whether the software system has to be continuously available or not during this evolution.
	Activeness	refers to reactiveness (changes are driven externally) or proactiveness (the system autonomously drives changes to itself) reaction of the system as the change happens.
	Openness	refers to an ability of the system for facilitating the inclusion of extensions.
	Safety	measures to what extend safety aspects are preserved as the change happens and if there are built-in provisions for preventing or restricting undesired behaviour at run-time. This dimension ranges between static and dynamic safety.
How	Degree of Automation	shows the degree of automation to perform the change, ranging from automated to partially automated and manual change support.
	Degree of Formality	ranges from ad-hoc way approaches to changes that are based on some underlying mathematical formalism.
	Change type	influences the manner in which the change is performed as either structural or semantic change.

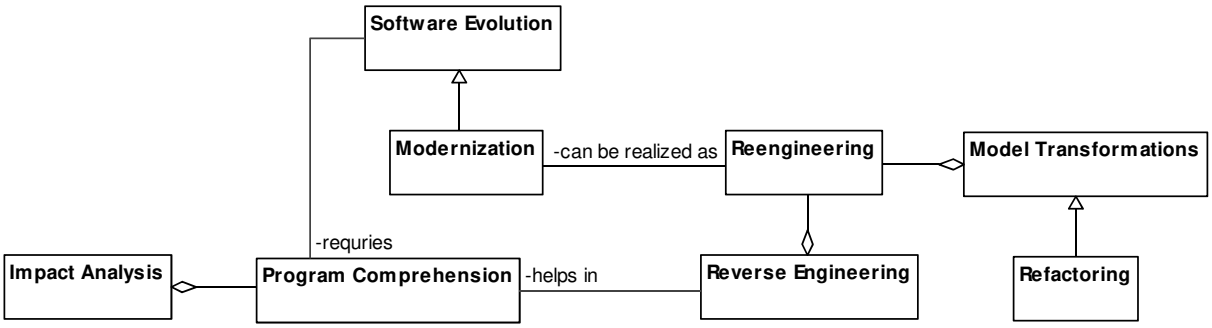

Figure 2.2: Relationships Among the Presented Software Change Concepts

requirements. Here, the changes are relatively extensive, but the original software still has business value that must be preserved. Reasons for modernization usually stem from legacy system brittleness, inflexibility, isolation, non-extensibility, and lack of openness [15].

A complete modernization effort should consider and support the following points of view [167]: (i) new functional and non-functional requirements, (ii) leading-edge technologies, devices, protocols, and standards, (iii) maximum reuse of existing assets, and (iv) cost and effort.

Modernization can be done in either black-box or white-box fashion. This depends on the required level of system understanding to support the modernization effort [196]. White-box modernization requires knowledge about internals of a system. In white-box modernization, the code is analyzed and understood in order to identify the change points and to restructure. This restructuring can be defined as transformations at a same relative abstraction level. In contrast, black-box modernization abandons the structure and data flows within the entity, by only focusing on the inputs and outputs, within an operating context, to understand the system interfaces. Therefore, the black-box approach is usually not as difficult as white-box modernization. Black-box modernization is often based on either wrapping or altering the composition of entities. Wrapping is a black-box modernization because only the interfaces are analyzed, and the system internals are ignored. Unfortunately, this solution is not always practical and often requires understanding the software module's internals, using white-box techniques [167]. This means the modernization might be black-box at one abstraction level, but white-box at a higher level of abstraction.

2.1.2 Software Reengineering

Reengineering, as a form of modernization, is the examination, analysis, and alteration of an existing software system for its reconstitution in a new form, and the subsequent implementation of the new form [40]. This may include modifications with respect to new requirements that are not met by the original system. The process typically encompasses a combination of other processes such as reverse engineering, re-documentation, restructuring, and translation. The goal is to understand the existing software at different abstraction levels and then to re-implement it to improve the system's functionality, performance or implementation. The objective is to maintain the existing functionality, and prepare for a functionality to be added later. As stated by the definition, reengineering consists of the examination and the alteration of a subject system.

As stated by Chikofsky and Cross [40], reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering. In this context, forward engineering is the process of moving from high-level abstractions and logical, implementation-independent designs, to the physical implementation of a system. The adjective forward in forward engineering is mainly used to distinguish classic software engineering process from reverse and reengineering. The main difference between (forward) engineering and reengineering is that reengineering starts from an existing implementation. Consequently, for every change to a system, the reengineer must evaluate whether (parts of) the system need to be restructured (or refactored [65]) or if they should be implemented from scratch. Here, restructuring generally refers to source code translation, but it may also entail transformations at the design level.

Although reengineering is not intended to be used for enhancing the functionality of an existing system, it is often used in preparation for enhancement. Reengineering specifies the characteristics of the existing system, which can be compared to the specifications of the desired system. The reengineered target system can be built to easily facilitate the enhancements. For example, if the desired functional improvement to a legacy system is dependant on object-oriented properties, the system can be reengineered to support object-orientation. This reengineering process does not introduce a new functionality in software and preserve its behaviour, but it allows future modernization tasks that are object-oriented dependant.

As a part of this research work, we will propose a reengineering process for modernizing software towards self-adaptivity, in which the reverse engineering activity basically maps to the act of analyzing and understanding the current software in terms of current adaptability, and the forward engineering activity is supported by the set of program transformations to increase software adaptability.

2.1.3 Reverse Engineering

Chikofsky and Cross [40] define software reverse engineering as "the process of analyzing a subject system to identify the systems components and their inter-relationships and create representations of the system in another form or at a higher level of abstraction." Reverse engineering is a common practice for comprehending software and a process of analyzing software in order to: (i) investigate the systems components and their relationships, (ii) extract information from source code, and (iii) create abstractions of the system that describe the underlying system structure and behaviour.

Reverse engineering has two main activities [34]: (i) analyzing software artifacts (including analyzing the evolution of software artifacts across their revisions), and (ii) providing software models that are understandable by humans. Software analysis is usually performed by tools that take software artifacts as input and extract information that is relevant to reverse engineering tasks. The analyzers can extract information using static analysis, dynamic analysis or a combination of both. The software models are basically abstract views of software. Several representation paradigms for software models are available in the literature [58], including logics and logical databases, sets and relations, relational databases, and graphs. All these different forms of code representation have their advantages and disadvantages. However, they should be able to query the information base containing facts populated by analyzers, and to abstract low-level artifacts and reconstruct high-level information.

During the last decade, various supporting techniques and tools are developed for reverse engineering. Canfora *et al.* [34] provide a list of common and popular tools for reverse engineering. Such tools and techniques help to: (i) extract facts from source code/binaries, (ii) investigate execution traces or historical data, (iii) query the extracted facts, and (iv) build high-level views for humans.

2.1.4 Program Comprehension

Program comprehension (program understanding) is the process of acquiring knowledge about software. This knowledge plays a crucial role in software evolution, as software must be sufficiently understood before it can be properly modified. Program comprehension is a long lived concept in software engineering. Numerous theories and approaches towards program comprehension are available in the literature [45, 156, 195], and are supported by various comprehension models [194]. These models can be viewed from different perspectives [156]. Two main dimensions for categorizing program comprehension techniques are:

- *Systematic vs. As-needed:* In a systematic approach the maintainer examines the entire program and works out the interactions between various modules [117]. This is completed before any attempt is made to modify the program. In contrast, in the as-needed strategy the maintainer attempts to minimize the amount of study prior to making a modification. Thus, the maintainer tries to locate the section of the program which needs to be modified and then commences the modification.

- *Top-down vs. Bottom-up:* This classification is proposed by Brooks based on the hypothesis of a mapping between the problem-domain and the solution-domain [25]. Brooks argues that domain knowledge is an important factor in program comprehension. He argues that the developer produces these mappings and the maintainer has to reconstruct them. This duality is described as top-down and bottom-up approaches. In top-down approach, the comprehension is achieved by setting a certain hypotheses and then confirming or rejecting them based on evidence. The confirmed hypotheses are retained, becoming part of the program's comprehension, while rejected hypotheses are discarded. In contrast, bottom-up approach of program comprehension is based on understanding fundamental elements and piecing together the inferred knowledge to deduct new knowledge by combining base elements into rather larger elements.

Impact Analysis

Impact analysis is a program comprehension activity of identifying the potential consequences, including side effects and ripple effects, of a change, or estimating what needs to be modified in order to accomplish a change before it has been made [19]. The basic principle underlying the need for impact analysis is that a small change in a software system may affect many other parts of the system, which is known as *ripple effect*. Two major class of techniques for impact analysis are [19]:

- *Dependency analysis*, which involves examining detailed dependency relationships among program entities. Data flow analysis, control flow analysis, and program slicing are common practices to support this analysis.

- *Traceability analysis*, which is the ability to trace between software artifacts generated and modified during the software product life cycle at various abstraction levels. For example, it can relate requirements with associated design components, or design models to source code.

Dependency analysis provides detailed information about dependencies in source code but does little for entities at other abstraction levels. Dependency analysis can be done in a static or dynamic manner and can target both structure and semantics of software [18, 95]. In contrast, traceability analysis techniques usually take into account the whole range of documents associated with software, including specifications and design. However, maintaining traceability information over a system's life cycle usually requires considerable effort, and thus is often slighted under the time pressures, which is associated with software development and maintenance in most cases. Techniques using information retrieval methods have been developed to recover traceability links, hypothesizing the consistency between high-level document terms and source code identifiers and comments [7].

2.1.5 Model Transformation

Models specify abstractions of real-world phenomena, where the abstraction level can vary and depends on the application domain. Such models are used for a variety of purposes; for example in software construction, abstract models serve as the basis for transformations resulting in the generation of (executable) source code. From this perspective, models are often used during the software development cycle and prior to runtime. Kleppe *et al.* [105] provided the following definition for transformation:

"A *transformation* is the automatic generation of a *target model* from a *source model*, according to a transformation definition. A *transformation definition* is a set of *transformation rules* that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language."

Considering the fact that source code is a model[126], this definition also includes *program transformation*, since an act of changing one program into another program is a model transformation. Model transformations can be applied for a very wide range of applications such as model driven engineering, reverse engineering, program optimization, code generation, migration, and many other applications in the domain of software maintenance and evolution. Due to the wide range of applications and perspectives, several taxonomies have been proposed for model transformations such as the ones proposed by Mens and Van Gorp [126], by Czarnecki and Helsen [46], and by Visser [192].

Transformations can be automated by using transformation tools and languages. When building modeling tools, one needs to model the structure and well-formedness rules of the

language in which the models are expressed. Such models are called metamodels [63]. Having a precise metamodel is a prerequisite for performing automated model transformations.

There is a multitude of tools and languages available for model transformation. Some of them are listed in [46]. Among the large list of tools and languages, the most widely used general-purpose language specification is Query/View/Transformation (QVT) [138]. The QVT specification was proposed by the Object Management Group. It consists of three languages: QVT Core, QVT Relations and QVT Operational Mappings. Another famous transformation tool is ATLAS Transformation Language (ATL) [96], which is generally used to express MDA-style [137] model transformations, based on explicit metamodel specifications [96]. ATL was created by INRIA[2] as an answer to the OMG[3]'s QVT language request for proposals. Besides MDA-style transformation languages, several graph transformation language are available (e.g., GrGen.NET [92], VIATRA2 [190], and GReTL [84]).

This research specifically benefits from GReTL [84], a graph-based general-purpose operational transformation language, for transforming models of software at runtime. GReTL operations are either specified in plain Java using the GReTL API or in a simple domain-specific language. GReTL follows the concept of incrementally constructing the target metamodel together with the target graph.

2.1.6 Refactoring

Program Refactoring is a class of program transformation which is used to improve software quality by restructuring it while preserving its behaviour. The term refactoring was originally introduced by William Opdyke [142]. The main idea is to redistribute classes, variables and methods across the class hierarchy in order to facilitate future adaptations and extensions [65].

Refactorings can be done manually or automated at different abstraction levels. Refactorings are widely used by developers to improve the quality of an object-oriented code, and are supported by software tools. Refactorings can be composed to form more complex transformations. Based on object-oriented refactorings, Tokuda and Batory [186] presented a pragmatic approach for high level object-oriented transformations to design patterns while preserving software behaviour. In another approach, for higher level transformations, Tahvildari *et al.* [182] use a catalogue of object-oriented metrics as an indicator in order to automatically detect where a particular meta-pattern can be applied to improve

[2]the Institute National de Recherche en Informatique et en Automatique
[3]Object Management Group

software quality using refactoring. In another work, Amoui *et al.* [3] develop a framework for automatic software evolution using design pattern transformations. The intelligent evolution engine is based on Genetic Algorithms and aims to increase software quality by mapping the problem into a multi-objective optimization problem where the objectives are set to software quality properties.

2.1.7 Software-Change Prediction

Change prediction plays a key role in effective planning of evolution and adaptation changes. Software change prediction can be performed for different purposes such as reverse engineering and cost estimation [71]. It often determines change-prone entities and change propagation patterns of a product [197]. A good prediction model can be used for future resource allocation and cost estimations, as well as specifying *where* and *when* maintenance/evolution tasks are "better" to start. This issue in large-scale systems is crucial for early detection of potential changes that may occur in future, and in turn reducing the costs and the risks of applying those changes.

Most change prediction approaches are based on the analysis of software history, and its evolution. The information is normally captured from software repositories and/or documentation [197]. Software change prediction techniques can be classified by two aspects: *what* and *how* they predict. Most of the research works related to the *what* aspect aim at [82]: (i) predicting the number of changes, bugs, or faults, or (ii) predicting the probability or rate of the change of software entities. On the other hand, the *how* aspect classifies prediction models as either stateless or stateful. The former relies only on the current state of a system, while the latter is based on past history rather than current state.

In addition, time-series prediction for software entities (such as software change/fault prediction) can be performed through two approaches: mathematical/statistical models [12, 76, 136], and AI/soft computing techniques [85, 102, 153, 169]. Most of these works address prediction of quality indicators such as maintainability and reliability.

Independent of prediction technique, predictors play an important role in the effectiveness of the whole process and quality of outcomes. Prior works have identified important predictors for software change prediction (e.g., Khoshgoftaar *et al.* [101], and Mockus *et al.* [130]). Suitable predictors should provide appropriate information with minimum possible prediction error along with a proper level of flexibility to tune prediction techniques.

2.2 Self-Adaptive Software

The trend of software evolution and maintenance is moving towards support for dynamic and runtime evolution [98]. Frequently changing user needs or high variability in the amount of available resources are only two examples of scenarios in which modern software must be able to reconfigure and continuously optimize itself at runtime.

The term "Self-adaptive software" was first introduced by the Defense Advanced Research Projects Agency (DARPA) as a software system that is able to evaluate its own behaviour, and change that behaviour when its evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible [108]. Another definition is given by Oreizy *et al.* [144]: "Self-adaptive software modifies its own behaviour in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation."

From the definitions above, one can conclude that self-adaptive software should contain the following characteristics:

- ability to observe changes in its operating environment[4].
- ability to detect and diagnose operating environment state transitions and evaluate its own behaviour.
- ability to alter its own behaviour in order to adapt itself to the new changes.
- support of dynamic behaviour. Its internal and/or external behaviour should be able to change intentionally and automatically.

2.2.1 Architectural Perspective

Self-adaptive software (SAS) must be *self-aware* in the sense that it knows at least a subset of its own structural and behavioural elements. This is an essential property of SAS, because changes to the state of software need to be observed by a controller that is able to respond to changes in the software's operating environment.

Computational reflection, as defined by Maes [122], is a solution for achieving the self-awareness property [198]. A *reflective system* is *causally connected* to a representation

[4]We assume that software is part of its operating environment.

of itself. If the system changes, its self-representation changes accordingly (*reification*). Conversely, changes in the self-representation result in changes to the underlying software (*reflection*).

In a *reflective architecture*, the part of software that specifies the business logic and interacts with the application domain is called the *base-layer*. It is causally connected to a *meta-layer*, which specifies the self-representation aspect of the system. Having a self-representative meta-layer allows software to reason about its own structure and behaviour in order to take required actions for its behaviour adjustment.

An important aspect of adaptation is the change granularity. This is often discussed in terms of two general categories of coarse-grained and fine-grained changes. A remarkable number of well-known studies in the literature address coarse-grained changes such as architectural changes, for instance [39, 145]. Although these changes can be highly effective, the problem is that they are hard to implement, test, and manage in complex large-scale distributed systems. This issue is much more difficult under unanticipated conditions. Fine-grained adaptation has been increasingly used, due to its lower cost, and also due to the advent of supporting technologies (e.g., dynamic aspect composition). Parameter adaptation changes are fine-grained, but some resource management actions and the composition of dynamic aspects also fall into this category. For instance, Grace *et al.* [74] use the fine-grained adaptation of aspect-oriented compositions by changing the cross-cutting concerns of a subset of entities [56, 77].

Adaptation approaches also differ in how base-layer and meta-layer are specified and communicate with each other and with other parts of the system. Basically these approaches can be divided into two main categories: (i) *internal approaches*, which treat and realize adaptation requirements as first-level functional properties of software, and (ii) *external approaches*, which separate adaptation concerns from other functional concerns of the system. Each approach has its own advantages and disadvantages. This section will introduce both approaches and argue their benefits and drawbacks.

Internal Approaches

Internal approaches interweave application and adaptation specifications. These approaches are mainly recognized as programming-based mechanisms for low-level adaptations. In support of internal approaches, there is a point of view of adaptive applications based on adaptive programming principles as an extension of object-oriented programming [114]: "A program should be designed so that the representation of an object can be changed within certain constraints without affecting the program at all." By this view, an adap-

tive program will be: "A generic process model parameterized by graph constraints which define compatible structural models (customizers) as parameters of the process model." This point of view is similar to reflection and meta-programming. The main advantages of internal approaches are: (i) testability, due to the expected behaviour of an adaptive software, (ii) applicability of formal methods approaches and model checking for increasing the reliability and robustness of the system, and (iii) better performance due to low-level language support.

The main drawbacks of the internal approaches can be summarized as: (i) limited support for dynamic adaptation, (ii) lack of scalability for large and complex systems, (iii) costly reengineering of adaptation requirements, and (iv) limited native support by programming languages and development tools.

Another interesting set of approaches are those which are considered internal at the design level, but may be realized as either internal or external at the code level [36]. However, due to the mentioned drawbacks of internal approaches, more attention is given to external approaches in practice.

External Approaches

External approaches use an external sub-system to control the adaptation of the software system, similar to classic feedback control systems [28], in which a system can either follow an open-loop or a closed-loop architecture. An open-loop (i.e., feed forward) system has an accurate model of its domain and adjusts the control output based on environment disturbances [141]. The feedback control and its variations, including adaptive control, inspire software engineers to employ the closed-loop architecture in software systems. The major merits of external approaches are: (i) adaptation requirements are separated from functional requirements, (ii) separation of adaptation logic and application business logic, (iii) increase of reusability, scalability, and maintainability, (iv) support of dynamic adaptation, and (v) applicability of artificial intelligence and control theory techniques.

The adaptation manager, as a controller, continuously runs four processes in a closed loop form: *monitoring*, *analyzing*, *planning*, and *executing*, as shown in Figure 2.3. Continuous execution of the manager shapes a control loop, which is known as *MAPE-loop* [87]. The loop is completed by connecting to the adaptable software through the provided sensor and effector interfaces. Here is a summary of each process depicted in Figure 2.3:

- **Monitoring** deals with collecting and correlating data from sensors and converts them to behavioural patterns and symptoms. The monitoring process can be realized by event correlation, threshold checking, and gauge monitoring.

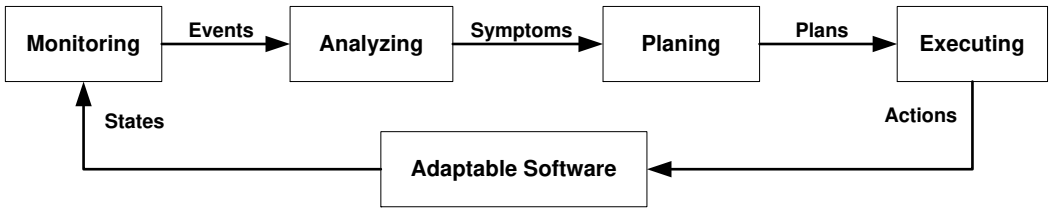

Figure 2.3: Architecture of an Adaptation Manager

- **Analyzing** is responsible for detecting the symptoms provided by the monitoring process and also the system's history to detect when a change needs to be applied.
- **Planing** determines what needs to be changed and what is the best way to make such a change. Planing is usually specified as *decision models* and supported by *decision-making algorithms*.
- **Executing** carries out actions planned by the deciding process. This includes managing non-primitive actions through workflows or mapping actions to dynamic adaptation technologies provided by adaptable software.

The next subsection briefly discusses techniques, mechanisms, and technologies that facilitate the development of adaptive software systems.

2.2.2 Enabling Technologies

There are several available techniques for probing and sensing software applications. Logging is likely the oldest and simplest technique to monitor software. The logs need to be filtered and analyzed to extract useful information. To assist this process, we can benefit from log analyzer tools. IBM's *Generic Log Adapter (GLA)* and the *Log Trace Analyzer (LTA)* [86] are examples of these type of tools. Besides logging, several enterprise standards and frameworks exist to add monitoring features into software. *Application Response Measurement (ARM)* is one of these standards [8]. *ARM* describes a method for integrating enterprise applications as manageable entities. It comes with an SDK for the Java and C languages, which enables developers to add sensors for measuring end-to-end parameters. The measurement parameters include application availability, performance, usage, and end-to-end transaction response time.

Profilers can also assist the monitoring process as either application sensors or middleware sensors. The *Java Virtual Machine Tools Interface (JVMTI)* is an example of

both an application and middleware profiler for Java applications [179]. *JVMTI* allows a program to monitor the applications running in the JVM. It also comes with a set of APIs for application level monitoring. Instrumenting the application for sensors can be done at compile time, as in gprof [75], or instrumentation codes can be added to the binary application as in *ATOM* [111, 174].

Another powerful management framework for Java is *Java Management eXtensions (JMX)*, which gives powerful facilities for both sensing and effecting [178]. The JMX technology, as a management framework included in the Java Development Kit (JDK), is widely used in Java environments for managing and monitoring resources (e.g., devices and applications) [178]. In JMX, each resource is instrumented by one Java object called Managed Bean (MBean). External applications can access MBeans via the JMX MBean server. MBeans can be accessed to observe the changes, and can be modified in order to control the application's behaviour [9].

In contrast to sensors, approaches for effectors are less general and more case specific. Effectors basically have the nature of action events. The adaptation manager (or the software itself in case of the internal approach) triggers action events, and software should follow some structural and behavioural infrastructures to support effectors as action events.

One solution to implement effectors is triggering action events by having appropriate tuning parameters in software and setting their values (e.g., the *Software Tuning Panels for Autonomic Control (STAC)* project, which aims to set tuning parameters as effectors into legacy software [22]). However, most of the solutions proposed for implementing effectors try to alter software behaviour by dynamically changing the application data and control flow (e.g., reflection [122], dynamic aspect weaving [151], and metaobject protocol [104]).

Aspect-Oriented Programming

Aspect orientation is used as a capable solution for developing self-adaptive software systems. Aspect-Oriented Programming (AOP) is a viable solution for modifying application's source code and executables as needed [103]. This is mainly due to the fact that separation of concerns (adaptation and system functionality) is an important factor in adaptive systems, which is essentially one of the the main advantages of aspect-oriented design. Existing AOP languages such as AspectJ [10] or AOP frameworks, like JBoss AOP [93] hide the implementation complexity of bytecode transformations involved. AOP concepts can be otherwise implemented using lower level bytecode manipulation libraries such as ASM [147], BCEL [184] or Javassist [170].

Generally, AOP frameworks define two things: a way to implement *crosscutting concerns* as code snippets, and a programmatic construct, a programming language or a set of tags, to specify how and where to apply those snippets of code. The terminology varies from technology to technology but in essence, the same concepts apply. Table 2.2 presents a brief introduction to AOP terminology, covering the most common AOP concepts.

Table 2.2: List of Common Concepts Used in Aspect-Oriented Programming

Concept	**Description**
Aspect	A modularization of a concern that cuts across multiple objects. Aspects can be implemented as regular classes implementing specific interfaces, or as simple methods with a predefined signature.
Join Point	A point during the execution of a program, such as the beginning or ending of a method execution.
Advice	An action taken by an aspect at a particular join point. Different types of advice include *around*, *before* and *after* advice.
Pointcut	A predicate that matches join points. An advice is associated with a pointcut expression and runs at any join point matched by the pointcut.
Weaving	The process of mixing aspects with other application types or objects to create an advised object.

Aspect-Oriented Programming (AOP), and more specifically dynamic AOP, facilitates encapsulating adaptation concerns in the form of aspects through dynamic runtime adaptation. It also helps to implement fine-grained adaptation actions at a level lower than components [77, 161]. Another approach is JAC [150], which uses a wrapping chain that can dynamically change existing or new join points. AOP can also be used for instrumenting sensors as in the IBM BtM (Build to Manage) tool [29].

Middleware

Developing adaptable software is not a straightforward task, because autonomic behaviours increase software complexity, and might conflict with other non-functional properties. One of the main goals of this thesis is to develop an infrastructure to make this easier. With respect to the generic reference architecture for autonomic systems proposed in [98], the adaptable software may refer to something as narrow as a single module, or as broad as application containers, middleware, or even operating systems. Therefore, we may consider an adaptable software as an application or service, in conjunction with its constituent

layers. This viewpoint enables us to take advantage of possible sensing and effecting facilities provided by the application's operating environment. As an example, the system load may be monitored from different views through either the operating system's API or state parameters (e.g., number of active sessions).

Therefore, there is a range of sensing and effecting mechanisms from case-specific application-level sensors and effectors to generic sensors and effectors provided by the application's operating environment. Having a manageable platform is a great benefit for the adaptable software, because high-level sensing and effecting mechanisms are usually more robust, and are realized by systematic solutions. A good platform that supports manageability should be able to provide a flexible and rich set of sensing and effecting mechanisms for all application layers. This can be achieved by providing management facilities, such as APIs, protocols, and standards. Thus, the appropriate platforms for developing manageable systems should have: (i) capabilities to observe and alter system states, (ii) load-time configurable modules, (iii) dynamically (runtime) changeable modules, (iv) well-defined module interfaces, and (v) mechanisms for single-point-of-change adjustment of state variables. Compared to direct application-level mechanisms, platform-level sensing and effecting mechanisms may add unwanted limitations (e.g., less accurate sensing data, and coarser-grain effectors) or put extra overhead on the system performance. However, generally speaking, platform-level adaptation exhibits less coupling between sensors and effectors and application's business logic.

Decision Making

The task of analyzing and deciding on the optimal adaptation action to execute, given the system state and context, is a planning problem [55]: "how do we find an optimal decision policy to perform adaptation actions for a given (hopefully complete and correct) model of the system state, a set of available adaptation actions, and a means of evaluating the result of adaptation actions?"

In general, planning and decision making in self-adaptive software is a special form of the Action Selection Problem (ASP) in autonomous agents [123] and robotics, especially behaviour-based robotics [157]. Similarities can be enumerated considering dynamism and uncertainty of the environment, as well as online decision-making. However, the problem in software is generally more complex than in robotics. This is mainly because software behaviour is not obeying physical rules, has more control variables and disturbances, and more complexity.

Several ideas from behaviour-based robotics seem promising for autonomic software.

The majority of existing autonomic managers use a model-based approach for adaptation. These systems normally utilize a sensor fusion model to capture events and update the model. The idea of behaviour-based robotics is to use distributed specific task-achieving modules, called behaviours, and to apply command fusion instead of sensor fusion (e.g., subsumption architecture [24]). By this mean, there is no need to develop, maintain and extend a coherent monolithic model for the autonomic element and its context. More related to the thesis approach, Salehie *et al.* use voting mechanism for decision making in autonomic systems [164].

Recently there has been an increased interest in using machine learning techniques for software adaptation. In [185], Tesauro presents a hybrid approach that allows *reinforcement learning* [180] to bootstrap from existing management policies, which substantially reduces learning time and costliness. He also demonstrates hybrid reinforcement learnings effectiveness in the context of a simple data-center prototype. Dowiling describes how reinforcement learning can be used to coordinate the autonomic components in a distributed environment [55]. His PhD thesis introduces K-Component, a framework to incorporate a decentralized set of agents which learn for adaptation. Also, Littman *et al.* have used reinforcement learning to address self-healing properties in the domain of network systems [118], claiming their system is able to learn to efficiently restore network connectivity after a failure.

2.2.3 Adaptation Frameworks

There are several successful projects related to runtime adaptation frameworks [165]. In this section, we focus on related projects, frameworks, and technologies, and briefly highlight their features.

The *Rainbow* framework [39] uses an abstract architectural model to monitor the runtime properties of an executing system. Certain components of Rainbow are reusable and the framework is able to perform model constraint evaluation and adaptation at runtime. The framework mainly targets architecture-level adaptation, where adaptive behaviour is achieved by composing components and connectors, instead of handling explicit behavioural models. Rainbow assumes that any target system provides system access hooks for sensing and that developers can use wrappers to add hooks to legacy systems for affecting purposes. Guidelines for introducing such changes are not a focus of this work.

The *DiVA* project [131] considers the design and runtime phases of adaptation. DiVA focuses on adaptation at the architectural level and is based on a solid four-dimensional modeling approach to SAS [64]. At design time, a base model and its variant architecture

models are created. These models include invariants and constraints, which can be used to validate adaptation rules. One of the main focuses of this work is to tackle the issue of explosion in the number of possible runtime system configurations (modes). The authors use aspect-oriented modeling to derive a set of modes by weaving aspects into an explicit model [132]. Finally, the generated modes are then used to adapt the system automatically.

Vogel and Giese [193] propose a model-driven approach which provides multiple architectural runtime models at different abstraction levels. For instance, more abstract models are derived from concrete source code models. Subsets of these abstract target models focus on specific adaptation concerns to simplify autonomic managers, and support separation of concerns among them. Like DiVA, this research also targets adaptation from an architectural point of view. Both of these approaches rely on enterprise and component-based architectures, such as JavaEE, which makes them less suitable for enabling adaptivity in legacy and standard applications. The presented implementation is based on the *mKernel* infrastructure [27], which offers comprehensive support for the management of Java enterprise applications.

TRAP/J [159] enables the addition of new adaptable behaviour to existing Java applications without requiring changes to the original source code or the virtual machine. TRAP/J utilizes behavioural reflection and AOP techniques. In this approach, software developers choose from a subset of existing classes and generate aspects and reflective classes to derive an *adapt-ready* version of their original software. TRAP/J focuses on the technological approach of how to make an existing Java application adaptable. It supports reification of method invocations, but not the reification of read/write operations on fields.

StarMX [9] is a generic configurable adaptation management framework. It separates management logic from application logic by using explicit sensing and effecting interfaces. StarMX has no dependency on application characteristics (e.g., architecture or environment) or adaptation concerns. Designed for Java-based systems, it incorporates *Java Management Extensions* (JMX) technology and is capable of integrating with various policy/rule engines.

AspectOpenCOM [74] is a reflection-based component framework. In contrast to the presented related work so far, this technology is focused on the *fine-grained adaptation* of already deployed aspects. AspectOpenCOM is implemented in Java, and the authors present a case study illustrating the performance benefits of their fine-grained approach to adaptation in the context of aspect composition. This research is not related to runtime models, but provides lower-level facilities to be used in AOP-based runtime adaptation in general.

2.3 Engineering Aspects of Self-Adaptive Software

To engineer a software system, a collection of *requirements* is needed to address various functional and non-functional characteristics of the desired system. According to Berry *et al.*, there are four levels of requirements engineering for and in dynamic adaptive systems [14]: (i) by humans, to describe the general behaviour of the system; (ii) by the system itself, to determine when it needs to adapt and what behaviour to adapt; (iii) by humans, to decide when, how, and where the system is going to adapt itself; and (iv) by humans, to carry out research about adaptation mechanisms.

In complementary research, Goldsby *et al.* propose new levels of requirements that reify the original levels to describe requirements modeling work done by adaptive software developers [72]. They identify four types of developers: (i) the system developer, (ii) the adaptation scenario developer, (iii) the adaptation infrastructure developer, and (iv) the adaptive software research community. Each level corresponds to the work of a different type of developer to construct goal models specifying their requirements.

As engineering SAS is a broad topic, this section only focuses on those aspects that are used in this thesis.

2.3.1 Specifying Adaptation Requirements

There are a number of research works on requirements specification of adaptive software. Sitou and Spanfelner present a model-based requirements engineering approach to systematically analyze and specify both system behaviour and adaptation behaviour, starting from customer and business needs [172]. In this research, the authors present the *RE-CAWAR* approach, which is a methodology that aims to augment the requirements engineering process for context aware and adaptive systems. The core of the methodology is an integrated model of the usage context. In another work, Salifu *et al.* present a problem oriented approach to represent and reason about contextual variability, and assess its impact on adaptation requirements. This approach extracts and specifies concerns about sensing and effecting behaviours that can detect changes and adapt in response. They encode monitoring and switching problems into propositional logic constraints [166].

Regarding formal approaches, a survey by Bradbury *et al.* [21] describes numerous research efforts to formally specify dynamically adaptive programs. Graph-based approaches model the dynamic architecture of adaptive programs, as graph transformations (e.g., Taentzer *et al.* [181]). A few efforts have formally specified the behavioural changes of adaptive programs, including those that use process algebras to specify the behaviour of

adaptive programs (e.g., Allen *et al.* [1], Kramer and Magee [107], and Canal *et al.* [33]). In another work, Zang and Cheng use temporal logic to specify adaptive program semantics [200]. They model an adaptive software as the composition of a finite number of steady-state programs and the adaptations among these programs. It assumes that adaptive properties of each program have already been specified with a Linear Temporal Logic (LTL) formula. They introduce the *Adapt operator-extended LTL (A-LTL)*, an extension to LTL for specifying an adaptation from one program to another. These specifications are further specified with the KAOS methodology [48] to provide a graphical wrapper to the formal *A-LTL* specifications of the semantics, in order to achieve a goal-oriented specification [26]. Another research by the same authors introduces an approach to create formal models for the behaviour of adaptive programs based on high-level requirements, and to verify and validate the adaptive models [199]. They have also described how to use these models to generate executable adaptive programs.

The adaptation manager is described by several processes. Although these processes are named differently in the literature, they basically refer to roughly similar behaviours towards adaptation (e.g., see [51] [165]). We can classify adaptation specifications, for each process of the MAPE-loop (as illustrated in Figure 2.3):

- *Monitoring specifications* describe the association between sensors and entities that hold adaptation related data. These specifications deal with *what, when, and how attributes should be collected from sensors?* The sensed value may lie in a different space from that attribute, in which case a transducer module may be required to transform attribute values Figure 3.2. Kokar *et al.* [106] also mention a similar module, called QoS, that uses the feedback-control architecture in the SAS domain.

- *Analyzing specifications* describe the links between attributes and goals. In fact, the analyzing process defines the semantics of error-level calculations in the feedback-control architecture. The analyzing process addresses the question of *how much has the adaptable software deviated from its goals?* Depending on the goals, designers may use simple threshold functions or goal analysis techniques to realize analyzing specifications.

- *Planning specifications* describe the core part of the adaptation logic (i.e., the control law). Theses specifications relate to error levels in classic control systems [141]; that is the difference between attributes and goals and to actions. There are a number of ways to express adaptation logic, such as state charts, Petri nets, or temporal logic.

- *Executing specifications* are related only to the action metamodel and describe the association between effectors and actions. For both monitoring and executing specifi-

cations, the relationships between sensors and actions are considered, for the purpose of determining a particular state, and defining new actions, respectively.

The above four specifications tell apart the adaptation mechanism. Numerous techniques and algorithms can be employed for the design and implementation of adaptation mechanisms [165]. Lightstone discusses a few of the foundational techniques for solving self-management problems, such as expert systems, tradeoff elimination, static and online optimization, control theory, and correlation modeling [116].

Self-* Properties

SAS adapt itself to its operating environment according to its adaptation specification. Adaptivity properties are often known as self-* properties [165]. One of the well-known sets of self-* properties are self-configuring, self-healing, self-optimizing, self-protecting, and four other underlying properties of self-awareness, context-awareness, openness, and anticipatory. Self-* properties in conjunction with other Non-Functional Requirementss (NFRs) shape a goal model for software adaptability.

Each self-* goal is an aggregation of other goals. Self-* properties, as goals of self-adaptive software, are highly related to quality factors of software and especially to well-known non-functional requirements such as usability, reliability, supportability, and maintainability. For analyzing the relationships among self-* properties and NFRs, IBM proposed a set of self-* properties called self-CHOPs, along with their relations to quality factors in the ISO9126 quality model. This set consists of the aforementioned eight major and minor properties [67]. The relationships between these properties and quality factors is depicted in Figure 2.4.

In fact, most of the existing self-* properties, including self-CHOPs, can be expressed as NFRs. This is mainly because these properties strongly address non-functional concerns of a software system. Self-healing addresses fault/failure, self-protecting deals with security, and self-optimizing mainly relates to performance. Self-configuring is basically an underlying property of other properties, but generally addresses context changes, which are mostly non-functional. However, properties may contradict each other and in many scenarios we cannot find an optimal solution that satisfies all properties to their best. In these scenarios, there is a need of a multi-objective approach to find a pareto optimal solution.

There is an extensive body of research for modeling NFRs as goals, mainly softgoals. For example, a NFR framework [41] gives a systematic approach to build and use Softgoal

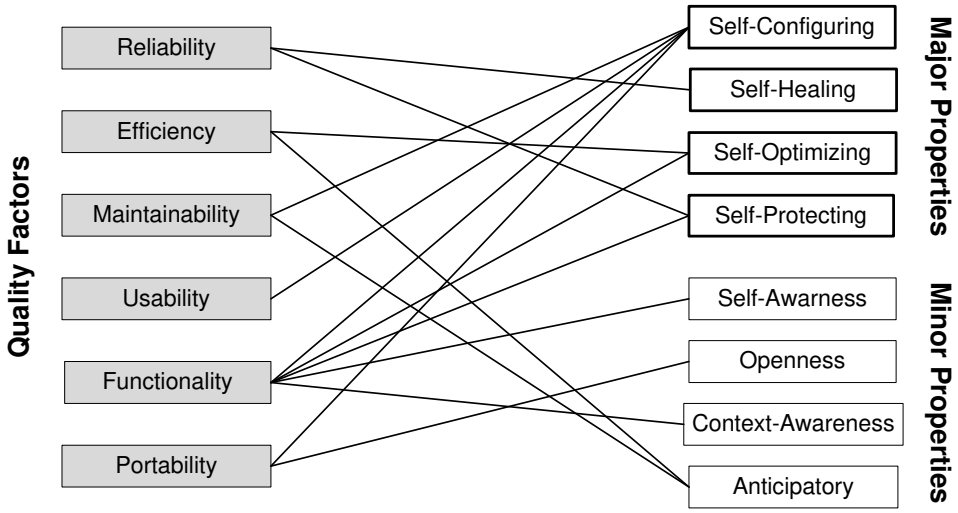

Figure 2.4: Relationships Between Self-* Properties and Quality Factors [162]

Interdependency Graphs (SIG), which can be used for goal modeling in self-adaptive software [110]. Based on SIG model, Subramanian and Chung [176] propose a treat software adaptability requirements as a goal to be achieved during development.

Software Adaptability

Software adaptability, as the main non-functional property of an adaptable software, is the ability of an application to alter its behaviour according to changes in its environment. This change can be either controlled by an external agent (another software or a manager) or by the software itself. In addition, both internal and external approaches require a mechanism in order to detect any state transitions in their operating environment. For example, Subramanian and Chung model software adaptability as NFR [176]. Through this approach, consideration of design alternatives, analysis of tradeoffs, and rationalization of design decisions are all carried out in relation to the stated goals. They also provide a tool to support developing adaptable architectures using NFR softgoals [177] as a part of a POMSAA *(Process-Oriented Metrics for Software Architecture Adaptability)* framework [42]. POMSAA aims to provide numeric scores representing the adaptability of a software architecture as well as the intuitions behind these scores, utilizing SIG.

Treating software adaptability as a NFR requires an evaluation technique to analyze the software for possible flaws against adaptability. Subramanian and Chung introduce two

high level metrics to measure the adaptability of software [175]. They define an *Element Adaptability Index (EAI)* for each software unit. *EAI* is set to 1 for adaptable elements and 0 for non-adaptable elements. *EAI* can be measured at different levels of software granularity. Based on *EAI*, the two metrics of *Architecture Adaptability Index (AAI)* and *Software Adaptability Index (SAI)* are calculated. In another work, Liu and Wang present two more metrics to evaluate the degree of adaptability [119]. These two metrics are mainly based on impact analysis of the whole architecture or architecture elements under a change requirement or adaptability scenario. These two metrics are *Impact On the Software Architecture (IOSA)* and *Adaptability Degree of Software Architecture (ADSA)*. Recently Tarvainen introduced *Adaptability Evaluation Method (AEM)* [183], that provides a structural adaptability evaluation method at the architecture level. Measuring *AEM* can help to improve architecture and decision making for alternate candidate architectures.

2.3.2 Design-Space Exploration

There is a spectrum of how adaptation takes place in self-adaptive software from static evolution to totally dynamic evolution [144]. For instance, one solution to static evolution for the purpose of runtime adaptation is having conditional expressions that combine the *adaptation specification* with the *application specification*. This approach provides limited but robust adaptation, since adaptation requirements are considered as software requirements. In addition, static adaptation may be realized by extending existing programming languages/systems [20, 83], or defining new adaptation languages [56].

On the other side of the spectrum are the dynamic techniques. These techniques can handle unforseen changes in the operating environment and provide a clearer separation of software-adaptation concerns. *Dynamic adaptation* is also categorized as *weak* or *strong* or even a combination of both [125, 163]. Weak adaptation or parameter adaptation involves changing the software parameters and variables to alter the program behaviour, while strong adaptation deals with more extensive changes in software, like changing algorithmic or structural parts of the software system. Dynamic adaptation can also be either fine-grained or coarse-grained [74]. The finer the adaptation, the lesser the required change in the software in order to alter its behaviour. However, there is always a trade off between cost, performance, and quality. Although fine-grained adaptation appears to be more powerful, it might be too expensive to have a fully dynamic fine-grain adaptation. Moreover, in many cases, coarse-grain adaptation (*e.g.* add, remove or substitute system components) is sufficient.

2.3.3 Retrofitting Adaptivity into Legacy Systems

For incorporating self-adaptivity into a legacy software system, there is a need for a evolutionary process for modernizing existing software into self-adaptive software as a three-step procedure: (i) preparing adaptable software by evolving the original software, (ii) developing or preparing an adaptation manager to observe and control the adaptable software, and (iii) integrating the two subsystems that are developed in the previous steps. Throughout this process, we have to initially analyze software (program understanding), which includes evaluating the current software in terms of adaptability, and identifying missing or conflicting adaptability properties in software entities. Next, we have to find a set of potential solutions to detected problems. This step is followed by a set of transformations to prepare the adaptable software. Finally, the last step is a forward engineering process, which involves testing the adaptable software, integrating it with the adaptation manager, and deployment.

There are several attempts to renovate and retrofit legacy systems into adaptive systems. Kaiser *et al.* propose an approach for renovating legacy software to adaptive software by adding a closed loop software controller [148]. Their approach, *Kinesthetics eXtreme (KX)*, consists of adding sensors and effectors to reconfigure running software. It has a similar reference architecture to that of [98, 144]. The sensors and effectors are program specific, but they utilize generic adaptation engines. In another work, Merideth and Narasimhanwe investigate how to retrofit network applications to add support for autonomic reconfiguration [128]. They explore how to retrofit pre-existing networked applications. As a case study, they retrofit a popular open-source intrusion detection system to enable it to reconfigure itself, using online program updates and information about its environment.

In another research, Muztaba Fuad *et al.* tries to inject self-organizing and self-healing properties into Java programs [146]. This approach is limited to Java byte-code transformations. The byte code is analyzed and additional code is injected to automatically recover from failures such as network or processor failure. The transformations are basically try-catch code wrappers to control application flow in case of failure. It also takes care of state variables by saving them at specified application checkpoints.

Another simple and operational approach is *STAC* by Dancy and Cordy [47]. *STAC* is an automatic transformation tool for re-architecting legacy software systems in order to facilitate autonomic control. It transforms software by exposing controllable variables in an external management control panel. The supporting transformations are developed using the TXL [44] tool for the Java language. The most recent update of the *STAC* project automates the discovery of software tuning parameters [22]. In this phase, the authors

create a catalogue of different types of tuning parameters in a set of open source software and organize them into a taxonomy. For each member of the taxonomy, they identify a source code pattern, used to find similar tuning parameters in other software.

2.4 Summary

This chapter presented a comprehensive review of supporting concepts for this thesis in the context of software adaptation. We also review the state of the art of concept and the related research works. The concepts were *software evolution*, *self-adaptive software systems*, and *engineering aspects of self-adaptive software*.

Upcoming chapters will discuss how the developed concepts, frameworks, and process models utilizes the discussed concepts and the state of the art techniques to tackle the problem of evolving software towards adaptivity in a unified manner.

Chapter 3

Conceptual Modeling of Self-Adaptive Software

"*Optimism is an occupational hazard of programming: feedback is the treatment.*"
–Kent Beck

The use of simplified abstract models through finding generic solutions and decisions is a common general strategy for dealing with complexity in software engineering [120]. Models specify abstractions of real-world phenomena, the abstraction level can vary and depends on the application. Models are used for a variety of purposes and are used during different stages of the software life-cycle. For example, in software construction, abstract models serve as the basis for software design and development. During software evolution, appropriate models of software can assist in understanding the software, and serve as the basis for model transformations that result in the generation of the evolved software.

One of the main challenges in engineering self-adaptive software systems is defining the problem space, and ultimately, setting the scope of a software solution based on an understanding of what will be built, as well as domain knowledge and a conceptualization of how information will flow through the solution. Moreover, maintaining self-adaptive software or migrating non-adaptive software systems to behave in an adaptive manner demand a comprehensive understanding of the internals of the software at hand. Such an understanding can be obtained with the help of appropriate domain-specific models of the software. Difficulties arise especially in the area of *model-centric runtime adaptation*, where models of software need to be generated, manipulated, and managed at runtime [69, 131].

As discussed in the following chapter, providing conceptual models of the problem space of self-adaptive software is essential to understanding the required solution, and how adaptation changes must be implemented and positioned in self-adaptive software.

3.1 Conceptualization

A software system interacts with an abstract world and affects it by changing some phenomena [89]. A phenomenon is any observable occurrence in the world that is shared between the software system and its domain. Figure 3.1 presents a Venn diagram, representing an abstract model of the self-adaptive software (SAS) based on Jackson's conceptual view of phenomena and phenomenon sets [79, 89]. This diagram models each of the *application domain (AD)*, *adaptable software (AS)*, and *adaptation manager (AM)* as a phenomenon set. The *AD*, *AS*, and *AM* abbreviations will be used through the rest of this chapter.

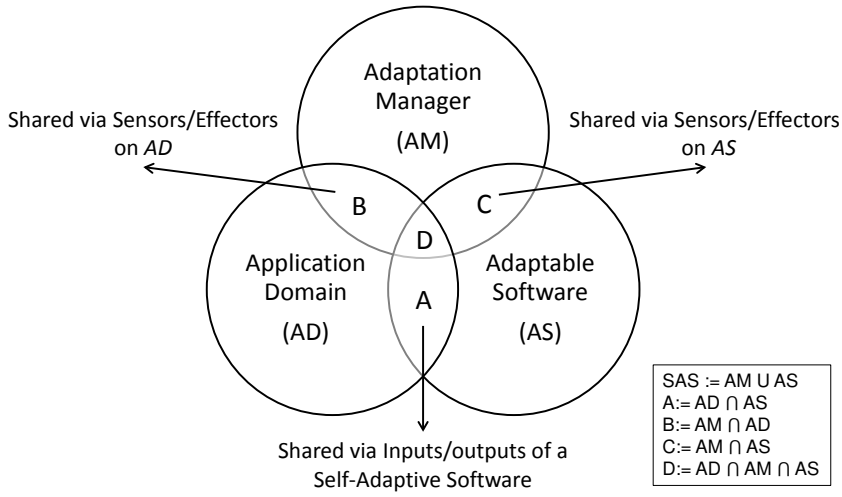

Figure 3.1: Phenomenon Sets in a Self-adaptive Software System

In Figure 3.1, the intersections of phenomenon sets correspond to the sets of shared phenomena between these entities. There are four intersection sets corresponding to the shared phenomena. Set A contains the shared phenomena between AD and AS that are shared through the input and output interfaces of the SAS. Set B contains shared phenomena sensed from and effected to the application domain. Set C are phenomena shared via sensor and effector interfaces exposed by the adaptable software. Finally, set D contains those phenomena that are shared between AD and AS and are exposed to

AM. Phenomena set D is especially important in the feedback-control architecture model, where the adaptation manager (controller) is dependant on the system's output.

The problem space of the SAS can be partitioned into two spaces: *adaptation* and *adaptability*. In the former space, the AS operating in its AD is controlled by the system (i.e., AM). In the latter space, the system (i.e., AS) interacts with the AM and AD. This explicit partitioning of problem spaces leads us to the architecture of a control-based self-adaptive software, which is composed of an explicit external *adaptation manager* (i.e., controller) and an *adaptable software* (i.e., controlled plant).

The possibility of generalizing and modeling SAS as a control system is also beneficial in the conceptual modeling of SAS, as many models and formalisms are available for various types of control systems. Hence, we create a mapping from control theory concepts to SAS problem spaces, following the model of a feedback control system, as illustrated in Figure 3.2[1].

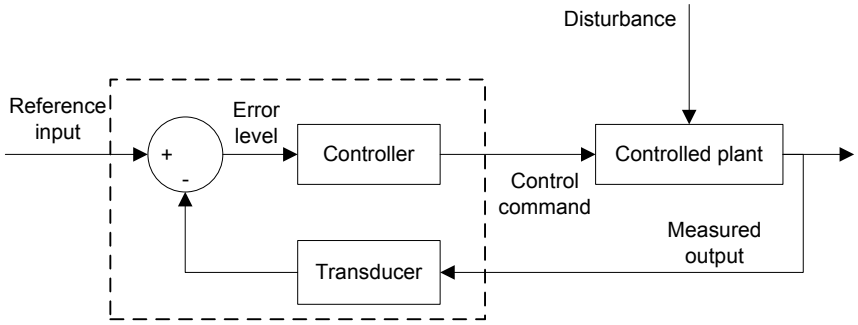

Figure 3.2: A Typical Feedback Control Architecture [141]

The result of this mapping is listed in Table 3.1. The first column of Table 3.1 includes the entities in a feedback control system with an external controller. The corresponding terms in self-adaptive software are listed in the two right columns for each of the adaptation and adaptability problem spaces. In this conceptual mapping, the adaptation manager covers the entities in the dotted box shown in Figure 3.2; that is, the controller, the transducer and the error evaluator. The reference input maps to the *adaptation goal*, which is used to determine the *Error level*. *Error level* is a significant property in the feedback loop and represents a the threshold of error (deviation from goals) to take an (adaptive) action, known as *goal denial level* in adaptation space. The main AM inputs include

[1]This model represents the basic feedback control system. More complex control system models used in software (e.g., as an adaptive control system) are identified by Kokar *et al.* [106]

Table 3.1: Mapping Control Theory Concepts to Self-Adaptive Software Problem Spaces

Control theory	**Adaptation Space**	**Adaptability Space**
Controlled Plant	Adaptable Software	System
Controller	System	Adaptation Manager
Transducer	Data Processors	–
Environment	Adaptable Software + (Application Domain)	Application Domain
Disturbance	–	Change
Control Law	Adaptation Policy	–
Error Level	Goal denial level	–
Reference Input	Part of system input data (i.e., Adaptation Goal)	–
Measured Output	Part of system input data (i.e Domain Attribute)	Part of system output data (via Sensor Interface)
Control Command	System output data (i.e Adaptation Action)	Part of system input data (via Effector Interface)

the adaptation goals and the measured output from the *AS*. Moreover, the disturbance input might be also provided for the adaptation manager, which is basically the change in the operating environment of *AS*. Using disturbance input data essentially adds the feed-forward property to the model. A part of the *AS*, according to adaptation requirements, needs to be sensed as *measured output*. *Adaptation logic* is the main part of behaviour specifications for the *AM*, and it is similar to control law.

Regarding system inputs, the input data includes *reference input* and *measured output*. The first two sets are the inputs and outputs of the software, which are transformed from monitored and controlled variables in the adaptable software and its environment. As noted before, the important point is that in a self-adaptive software system, the adaptable software and its environment together form the environment of the adaptation manager. On the other hand, the inputs for the adaptable software are an aggregation of the inputs from the *AD* and the *adaptation actions* from the *AM*.

For a controller, the output set includes control commands for the actuators. In self-adaptive software, the output data consists of selected actions for specific effectors. A simple example of such an action is increasing or decreasing a variable. These actions are ways that the adaptation manager can change the controlled variables.

3.2 Adaptation Requirements

The *requirements* of a system describe what is expected from the system in terms of its effects on its environment [89]. Functional requirements specify the desired changes to those phenomena of the AD that are shared with the system through input and output interfaces. The system observes a change in the AD's phenomena via an input interface, and in turn changes some shared phenomena via an output interface. In contrast, non-functional requirements specify changes to phenomena that are either (i) part of the AD, but are not shared phenomena and are changed as a side-effect of the application's behaviour (e.g., performance requirements and quality of service requirements), or (ii) part of other domains (e.g., business domain, test domain, and process domain).

Based on these requirements, engineers analyze the boundaries of a software system with its environment and develop specifications. *Specifications* describe the system to be built, which must fit and interact with its environment. In [89], Jackson describes the relation between requirements, specifications, and application domain as:

$$D, S \vdash R \tag{3.1}$$

Here, D denotes the underlying properties of the application domain and S is the properties of the system to be built. The argument is that the operation of a system with properties S in a domain with properties D will result in properties R. Now, if we consider R as the required properties identified by the requirements, then the problem is to specify a system with properties S (Specifications) in a way that if it operates in a domain with properties D, it would satisfy R.

The next step is to build the specified system, considering its platform and operating environment. Jackson describes the relationship between a software system and its specification as:

$$C, P \vdash S \tag{3.2}$$

where C denotes the properties of the computer, including its hardware description, operating system, and the semantics of the programming language in which the software is written, and P denotes properties of the software. Executing the software will result in properties S (i.e., the software's specifications). Here, the software engineering problem is to build a software system with properties P that satisfies the specification S, given an operating platform with properties C. In classic software engineering, we assume that the

domain (D) and the operating environment (C) are fixed, and the problem is to find an appropriate specification (S) and to build software (P) according to this specification.

In SAS, a class of requirements, known as *adaptation requirements (AR)*, captures what the system should do maintain quality at an acceptable level in almost all (even unforseen) situations. This means that the domain D and the operating environment C are not necessarily fixed and might change over time. Such situations may be because of external system attacks, new working environments, system failures, high system loads, or many other events that impact the system. The common characteristic of these situations is that we do not know what is exactly going to happen, or when it will happen. Adaptation requirements basically define system objectives in terms of phenomena in the original software's domain space (i.e., application domain), based on domain knowledge related to the presumed environment facts and the relationships among phenomena in the different problem spaces. In other words, adaptation requirements describe the adaptive behaviours of a system from the user's perspective. If adaptive behaviour has an impact on the output, then the adaptation requirements are considered functional. Otherwise they are considered non-functional.

Adaptation requirements are part of the system requirements ($R_{Adaptation} \subset R$). Recall the definition of SAS, as a system that is able to (i) recognize changes in its domain, (ii) determine the required changes to be made to itself based on changes in its domain, and (iii) apply changes to itself to generate an alternative system behaviour [176]. We notice that this definition refers to phenomena in the SAS that are to be affected by alternate behaviours of the system. In a control-based SAS, such phenomena are within the *AS*.

From the *AM*'s perspective, adaptation requirements can be treated as functional requirements for developing an AM as an independent sub-system, to monitor and control these phenomena in the *AS*. The four-variable model [2] proposed by Parnas and Madey [149] can be used to express this perspective. Figure 3.3 presents a mapping from the original four-variable model to the adaptation space. The adaptation space highlights key entities of the adaptation problem space and AM as its solution software system. In this mapping, monitored variables are a set of phenomena in the AS and AD that are to be shared with the AM via sensors. Moreover, the output data are a set of phenomena in the AM that are to be shared with the AS via effectors.

On the other hand, AM interacts with its world ($AD \cup AS$) via sensor and effector interfaces on the AS and the AD, as denoted by phenomena sets B and C in Figure 3.1,

[2]Four variable sets are: monitored variables, input data, output data, and controlled variables.

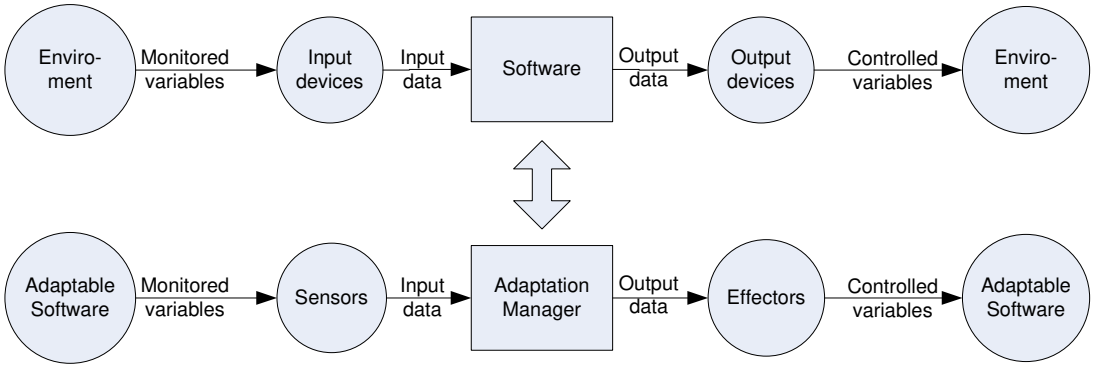

Figure 3.3: The Four Variable Model (Adapted from Parnas and Madey [149]) and Its Mapping to the *adaptation space*

respectively. The desired behaviour of an AM in its interaction with this joined domain can be formulated as:

$$(AD \cup AS), S_{Adaptation} \vdash R_{Adaptation} \tag{3.3}$$

However, the joined application domain ($AD \cup AS$) includes an unexplored shared phenomena set ($C - B$), from Figure 3.1, which holds part of the knowledge required to construct the AM. This set holds phenomena that are required to be exposed and shared via sensors and effectors applied to the AS. There are two ways to model this phenomena set: (i) considering sensors and effectors as part of the AS and capturing their properties as $S_{Adaptability}$, where $S_{Adaptability} \subset S_{AS}$, and (ii) considering them as part of the operating-environment properties of the AM, and capturing their properties as C_{S-E}, where $C_{S-E} \subset C_{AM}$. Here and through the rest of this thesis, we follow the first approach, because the second approach prevents us from specifying changes to be made to the original software, in order to make it adaptable in the case of migrating current systems towards SAS.

Assuming that a software system fully complies to its specification, we can substitute the system (program and computer) by its specification. Considering sensors and effectors as part of the AS and capture their properties as $S_{Adaptability}$, where $S_{Adaptability} \subset S_{AS}$, we can conclude that $S_{Software} \cup S_{Adaptability} = S_{AS}$. As a result, we can reformulate Equation 3.3 as:

$$AD, (S_{Software} \cup S_{Adaptability}), S_{Adaptation} \vdash R_{Adaptation} \tag{3.4}$$

If we completely decoupled sensors and effectors that are exclusively specified by $S_{Adaptability}$, we will have:

$$AD, S_{Software}, S_{Adaptability}, S_{Adaptation} \vdash R_{Adaptation} \tag{3.5}$$

and if *Software* is operational in the AD, we can rephrase Equation 3.5 as:

$$(AD \cup Software), S_{Adaptability}, S_{Adaptation} \vdash R_{Adaptation} \tag{3.6}$$

Therefore, AR implicitly holds two distinct sets of information: (i) requirements for sensor and effector interfaces to be provided by the AS (i.e., $S_{Adaptability}$), and (ii) functional requirements to construct the AM (i.e., $S_{Adaptation}$). This duality matches the adaptability and adaptation problems spaces as the former addresses adaptable software and how adaptability is specified, and the latter deals with the adaptation manager and how *adaptation specifications* are determined. We believe that both problem spaces (and their corresponding specifications) are equally important in the engineering process of SAS. Adaptation specifications describe how states and actions are defined and used to specify the AM's functionality. The *adaptability specifications* focus on specifying the required sensors and effectors from the AS's view.

However, a single, unified conceptual model of SAS is preferred as the spaces are not disjoint. Having such a unified view will enable system designers and developers to determine the boundaries of their subsystems and how the AS and the AM should interact with each other. As we will see in Chapter 5, this unified view is critically important for the problem of migrating current systems towards SAS. We present such a *unified conceptual model* of control-based self-adaptive software in Figure 3.4. This model captures the requirements of adaptability and adaptation problem spaces in two sets of specifications, including main entities of each space and their relationships.

In Figure 3.4, adaptation requirements are specified in a form of adaptation specifications and adaptability specifications. Adaptation specification describes functional requirements of the adaptation manager in a form of goals, actions, and attributes, and the relationship among these entities. Each attribute hold a shared phenomenon from the adaptability space that are shared via different interfaces (sensors) on the AS, its operating environment, or their interfaces (i.e., software's input/output interface). On the other hand, adaptability specification describe the required sensor and effector interfaces to be exposed by the AS. Here, we consider the sensors and the effectors part of the AS, but other sensors and effectors may also be provided by the environment that directly share *software output*, or *environment state* with the AM.

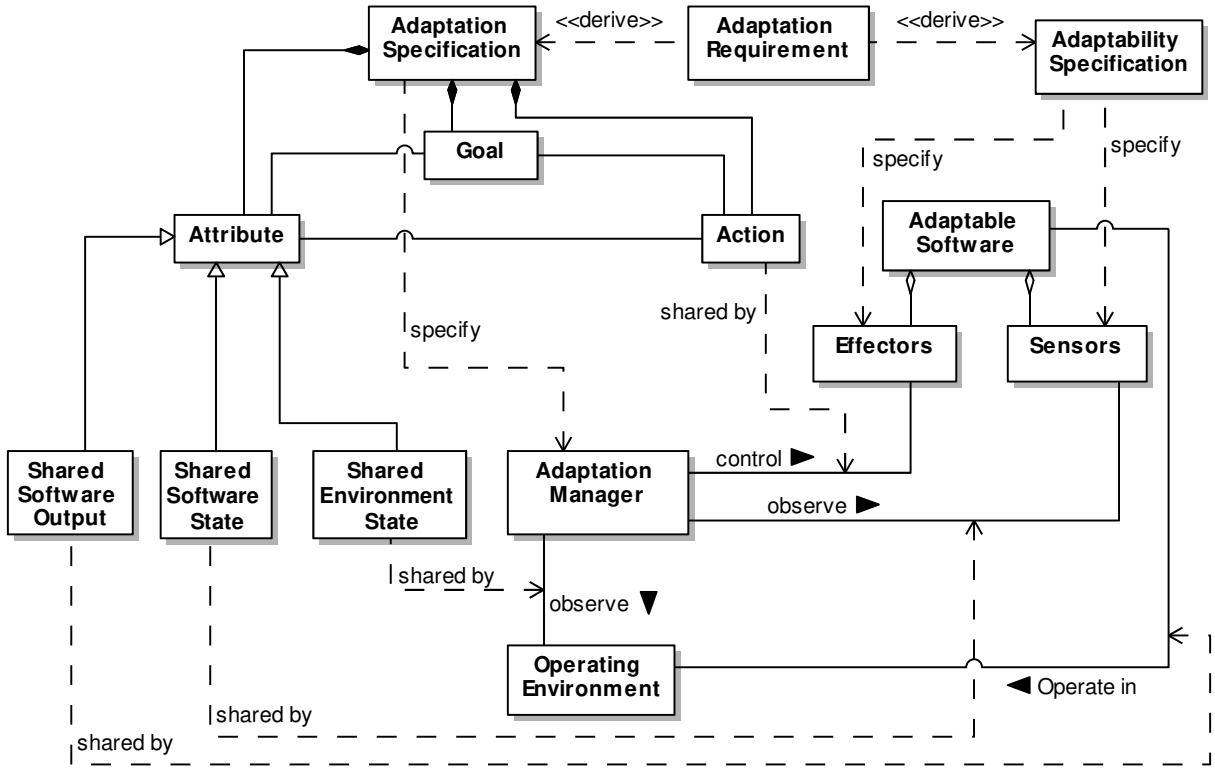

Figure 3.4: The Unified Conceptual Model for Controller-based Self-Adaptive Software Systems

In case where the current software is not adaptable and the solution demands an external adaptation manager, then the main problem is to derive adaptability specifications in such a way that balances the evolution cost of changing the AS and developing the AM, against the adaptation cost and effectiveness. In this sense, the synergy between the adaptability and adaptation problem and solution spaces can be treated as a co-evolution relationship; here, the problem is to come up with the fittest adaptability and adaptation specification models for maintaining model integrity and distilling the models, as abstractly shown in Figure 3.5. This figure basically shows how a change in one space can result in a change in other spaces. For example, a new adaptation policy may require a sensor data that is not currently available. This will affect the adaptability specification, as the adaptable software have is required to include the sensor. Adaptable software (in the solution space) will be changed to include the new sensor and accordingly the adaptation manger needs to be adjusted to use the new sensor.

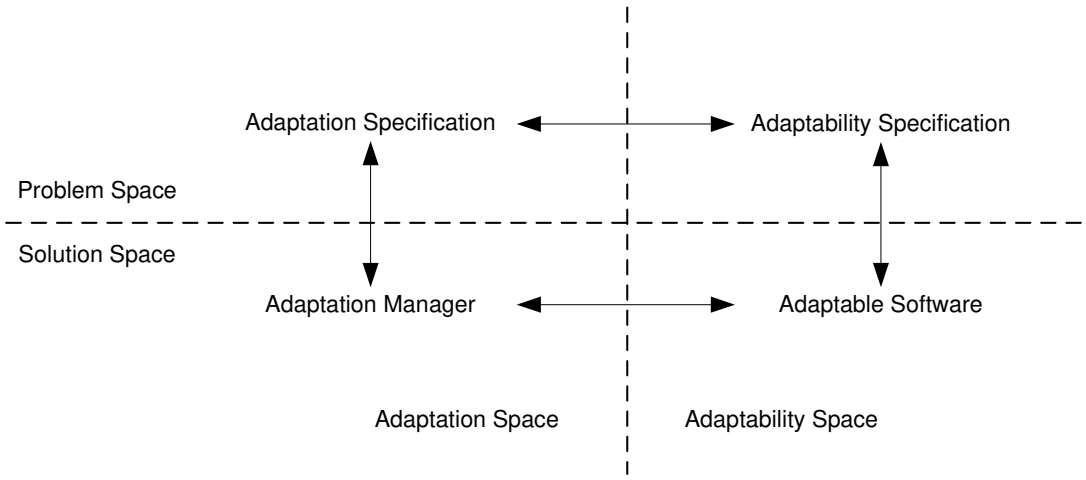

Figure 3.5: Coevolution of the Artifacts in the Conceptual Space of SAS

The application of adaptability and adaptation specifications is similar to the *specification* pattern introduced by Evans and Fowler [62]. The original pattern mainly serves as an analysis pattern that captures how people think about a domain, and as a design pattern suited to some system tasks. The following sections extend this view by elaborating these two specifications. Since specifications and design decisions impact each other in engineering SAS, whenever it is necessary, we note architectural and design considerations.

3.3 A Metamodel for Adaptation Specifications

An adaptation specification must cover two aspects: (i) a specification of interfaces to and from the adaptation manager, and (ii) a behavioural specification of the adaptation manager that associates inputs (i.e., adaptation actions and adaptation goals) to outputs (i.e., adaptation actions). Figure 3.6 illustrates the adaptation metamodel that can be used to specify the adaptation problem space. This metamodel includes domain attributes, adaptation actions, adaptation goals, and adaptation policies.

The rest of this section describes goals, attributes, and actions, as the key entities in the input and output sets, and the adaptation policies.

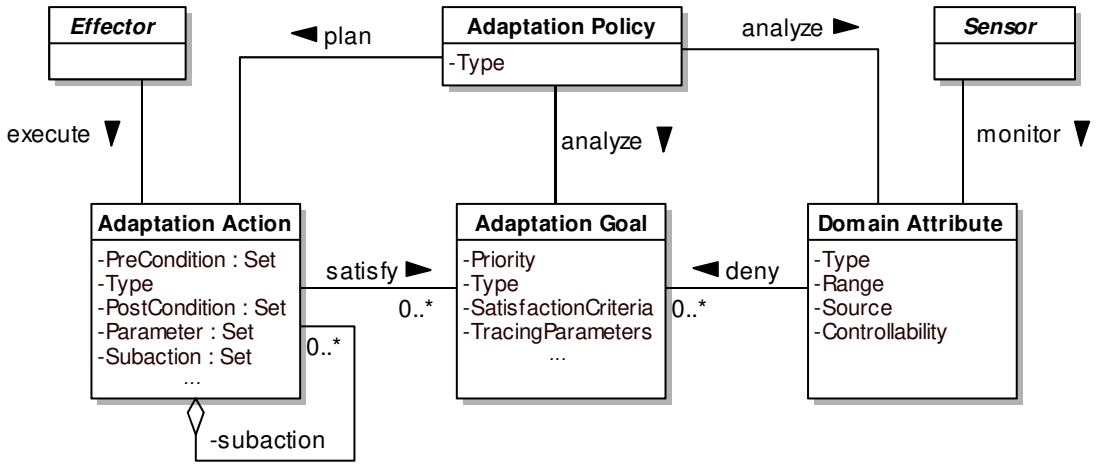

Figure 3.6: The Adaptation Metamodel

3.3.1 Adaptation Goals

Goals play the role of a reference for adjusting system behaviour. The use of goals in this context does not mean that we necessarily need to have a goal-oriented approach and should define explicit goal entities during development or at runtime. Recalling the equivalent concept in control theory, a goal generally means a reference input that serves as the basis for evaluation. The goal notion is necessary for specifying the error level (deviation from the goals). In this way, the conventional open-loop nature of the software's mapping of inputs to outputs, is changing to the closed-loop mapping of inputs to outputs, which takes an error level into account. Appropriate adaptation actions can vary based on the measured error level. For example, if the system load is slightly higher than the acceptable range, this is most probably due to higher user requests. However, if the load is much higher than the expected error level, it can be an indication of external system denial of service attack.

Adaptation goals are often linked to quality attributes such as performance and security [191, 162]. Kokar et al. [106] also point out that software output by itself is often not useful for the controller in the software domain, and needs to be transformed into a quality of service level; it is implied that the quality attributes of the adaptable software determine the error level.

This thesis does not address the elicitation or derivation of adaptation goals. Such goals are either derived from the adaptable software requirements or elicited from stakeholders.

In the former case, a goal may be the runtime presentation of an adaptable-software goal or a secondary goal defining conditions on an adaptable-software goal. For instance, a secondary goal can be defined to prevent the failure of another goal, or to insure its satisfaction for a specific period of time. Moreover, stakeholders might specify some adaptation goals, or quality goals might be defined for the adaptation manager (e.g., the adaptable software should continue its work if the adaptation manager fails).

Similar to goal-oriented requirements engineering [189] adaptation goals are not independent from each other and there may be several goal hierarchies for each goal. However, depending on how the adaptation manager needs to be operated, goal dependencies need to be refined and prioritized. For example, adaptation goals can shape a hierarchial model decomposed down from qualitative to quantitative goals. High-level quality goals (e.g., achieving an acceptable level of performance) are decomposed into low-level goals that are related directly to domain attributes (i.e., *leaf goals*). "Minimizing response time" and "having 80% resource utilization" are two examples of leaf goals.

In the proposed adaptation metamodel (see Figure 3.6), an adaptation goal has a *priority*, which represents the importance of a goal in relation to other adaptation goals. There are a number of classifications of adaptation goals [48, 135]. The choice of classification depends on the adaptation mechanism and how it utilizes goals and their models. A comprehensive goal taxonomy is defined by KAOS, which covers two dimensions [48]: (i) goal patterns: achieve, cease, maintain, avoid, and optimize; and (ii) goal categories: satisfaction goal, information goal, robustness goal, consistency goal, safety goal, and privacy goal.

Satisfaction criteria (aka fit criterion [189]) specify how the adaptation manager can determine whether a goal has been satisfied. Satisfaction criteria are associated with the link between goals and attributes. In the feedback control, an error is characterized by the difference between the reference input and measured output. Goal satisfaction or satisfaction level can define this error in SAS. In the simplest form, a goal can be a threshold, and the associated error can be the difference between the measured output and the threshold. For a non-leaf goal g_i, the satisfaction statement is based on the satisfaction of its subgoals, while for a leaf-goal, satisfaction depends only on the satisfaction criteria between the goal and its attribute(s).

Tracing parameters specify when and how often goal satisfaction should be evaluated. Such parameters may be different for each goal depending on the goal's probability of change, its priority, and the attributes related to the goal. For secondary goals, satisfaction criteria depend on other goals. For example, the failure of or the number of failures of a goal may deny a secondary goal. Secondary goals can be defined as awareness re-

quirements [173], adaptation goals in FLAGS [11], or secondary security goals [80]. The relationship between goals and actions is presented as the *GoalImpact* association class (discussed in Subsection 3.3.3).

3.3.2 Domain Attributes

Attributes are inputs to the adaptation manager from sensors. As Figure 3.4 illustrates, these attributes include measured outputs from the adaptable software, environment states, or internal states of the adaptable software. Attributes can be either linked to runtime models of the adaptable software (e.g., architecture model) or directly linked to sensors instrumented for adaptation.

Domain attributes include monitored variables of the adaptation manager. These attributes may come from the adaptable software or its context. In the former case, attributes can be either linked to runtime models of adaptable software (e.g., architecture model) or may be directly linked to sensors instrumented for adaptation. The attribute space may consist of models which include adaptable software attributes or context variables.

Domain attributes provide a source of data for creating various models to be used by the adaptation manager, such as configuration models, state models, and context models. These models use different combinations of self and context attributes to represent the system status. If each state is represented by a set of tuples, different spaces can be defined based on the category of attributes, such as a general state space.

Each attribute has a *type* (e.g., time) and a *source*, which can be from a component or model belonging to the adaptable software (e.g., failed/working state, and buffer length), or its environment (e.g., number of active users). The controllability field determines whether the attribute is under system control. We assume that an environment attribute cannot be controlled directly. However, it might be possible to indirectly change these attributes. For example, admission control actions can decrease the number of user requests to a web application.

The attribute may also have some meta information. The *sensing mode* (synchronous or asynchronous), *monitoring period*, and other required specifications are some of these parameters. For example, *importance* specifies how critical this attribute is for adaptation. Based on the level of awareness, the adaptation manager may need to monitor different sets of attributes. It can take any of the values in $\{critical, regular, optional\}$.

3.3.3 Adaptation Actions

An adaptation action is basically a transition from a source variant of adaptable software to a target variant. An adaptation action can be modeled in various ways; such a model at least includes the action's preconditions, where it will be applied, what it changes, and its impact on goals or attributes.

The impact of domain attributes and adaptation goals are formalized in the form of a precondition and a postcondition. A *precondition* is a condition or predicate that must always be true just prior to the execution of an action. It can be a specific condition under which the action is applicable, or a constraint attributed to effectors. Satisfying this condition is mandatory for executing an action. For example, video content should be available for users before applying a 'disable video' action. In a sense, a precondition is a domain precondition that is independent of the relationships between the action and other entities. A *postcondition* is a condition that must be true just after the execution of an action. A postcondition can be used for verification and validation purposes, in the form of assertions that reflect the expected impact of an action on other entities.

Adaptation actions can be simple or composite. A composite action can manipulate several effectors or perform some conditional sequences. Composite actions can be defined as transitions between these configurations or system states [189]. On the other hand, an adaptation change can be considered as an activity that is a composition of actions chained together in order to form plans. For example, restarting a component may be planned as a sequence of storing the component state, undeploying the component, redeploying it, and restoring its state. Actions can be parameterized too. Action *parameters* can be used to fine-tune adaptation changes.

Besides the above properties, additional information may be gathered and stored about actions. Examples include the history of action execution, the latest execution of an action, and the failure/success of previous executions.

3.3.4 Adaptation Policies

As discussed by Kephart and Walsh, an SAS matches the definition of rational agents [99]: "A rational agent is any entity that perceives and acts upon its environment, selecting actions that, on the basis of information from sensors and built-in knowledge, are expected to maximize the agents objective."

The behaviour of a rational agent is described by action, goal, or utility-function policies [99]. Policies compose elements of the behavioural specification of the adaptation

manager. Regardless of how policies are expressed, they are based upon the notions of states and actions [158]. Policies describe the relationship between the states and actions needed for a rational agent to achieve a goal. Selecting the adaptation actions depends on the desired adaptation goals and properties. A rational agent observes the current state, and performs an action accordingly to make a transition to a new state.

In general, for action-based (rule-based) policies, developing adaptation logic is as straightforward as coding required pairs of state-action policies that provide complete coverage of the state-action set. The state-action pair policies can be directly described with temporal [109] or propositional logic [97]. In contrast, goal-based and utility-function based polices are used in cases where specifying a complete state-action set is infeasible or costly; such cases arise for several reasons, such as: (i) a large state space, (ii) dynamic environments that may introduce new states, and (iii) a lack of knowledge regarding the effect of the selected action (it cannot be assured that the action contributes to achieving the desired goals). An intelligent adaptation manager can be used in these cases. It is also possible to infer the best action based on history (experience), for dynamically creating a mapping between states and actions. For example, an adaptation manager that uses *reinforcement learning* [180] can partially observe the state space, and accordingly select the best possible action based on its experience [185].

3.4 A Metamodel for Adaptability Specifications

Software adaptability consists of two dual properties: *observability* and *controllability*. Software is observable if its state can be determined by its output. Software can be partially observable if only part of its state space is observable. Moreover, software is controllable if it can be set to any desired state (from the current state) within a finite time. Hence, the mechanisms for supporting adaptability in software applications can be classified as the ones that enable observability and the ones that enable controllability. Therefore, $S_{adaptability}$ can be defined as observability and controllability properties added to the original software properties by introducing and exposing sensors and effectors.

The interaction between the AM and the AS in Figure 3.1, as a part of the AM's application domain, is through sensor and effector interfaces. Sensor and effector interfaces provide the required observability by exposing additional (direct) output that represents the software state, and the required controllability by exposing additional input that changes software state. The sensor interface serves as the AM input channel for observing any changes in the application, and the effector interface controls the application's behaviour

when needed. These are the same shared phenomena in the AS that are shared with the AM, and we are interested in specifying them via adaptability specifications.

If a perfect mapping from the AD to the AS exists, we can satisfy the AR by exclusively specifying the sensor and effector interfaces between the AS and the AM. Otherwise, we cannot completely satisfy the AR by solely effecting the AS. In the case of having a perfect mapping, we can redefine the AS as:

Given a set of ARs, a software system is considered as an AS if and only if it exposes the required monitoring data via its sensor interface, and exposes the required effecting operations via its effector interface.

To engineer the AS for a given AR, we can either develop a new software application from scratch, or retrofit a legacy software system to comply with the required adaptability specifications. The AR may change over time; a changed AR might require additional sensors and effectors, which can be added to the AS by a set of evolution and maintenance changes. In all cases, if the candidate software for self-adaptation is maintainable and reusable, it will be easier to change it and hence it can be retrofitted to an AS more easily.

Our approach to specify software adaptability is based on identifying a set of *sensing styles* and *effecting styles*. In this approach, each required sensor and effector in the AS is categorized as one or more styles. Each style can be realized by various techniques and enabling technologies. However, regardless of the techniques used, a style is composed of a set of abstract elements that we call *adaptability factors*. In addition, sensors and effectors can access and share state variables, too. These state variables are also part of the adaptability specifications and we capture them as adaptability factors.

Our approach for expressing adaptability specifications is based on investigations on sensing and effecting styles and their composing adaptability factors. Hence, our proposed adaptability specifications describes software adaptability in terms of the adaptability factors needed for enabling the required sensor and effector interfaces. The top-level view of the meta model for software adaptability is presented in Figure 3.7, which will be elaborated further throughout this section and be captured in Figure 3.10, Figure 3.11, and Figure 3.12 . In this metamodel, the adaptable software exhibits and exposes a set of sensors and effectors. Each sensor and effector is a realization of one or more sensing or effecting styles. The adaptable software software also has a set of state variables that can be monitored or controlled by the sensors and effectors (directly or indirectly). Each sensor and effector is composed of a set of adaptability factors. As we will discuss through the rest of this section, sensing and effecting styles in conjunction with their composing adaptability factors are the core elements of the metamodel for adaptability.

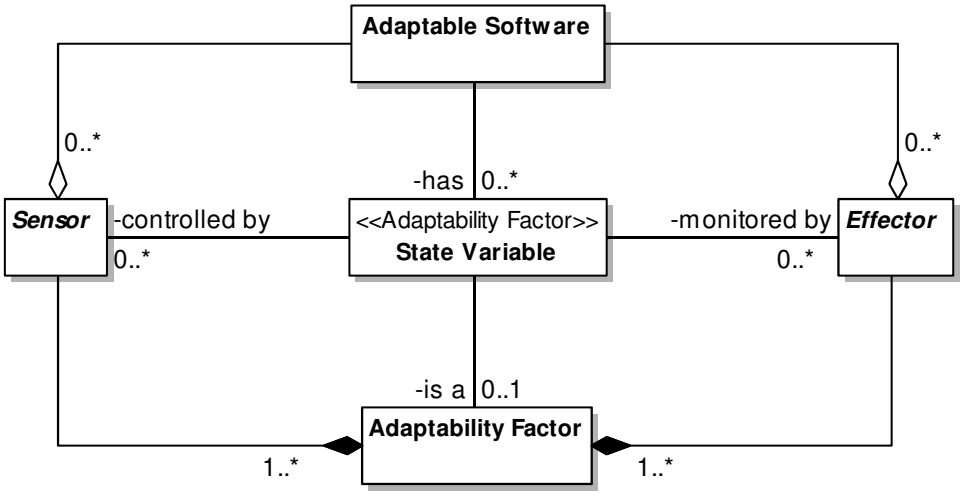

Figure 3.7: A Metamodel Representing Adaptability in Software Applications

3.4.1 State Variables

State variables play an essential role in adaptable software [133]. Some state variables represent the application's states, while others represent the operating environment's states (e.g., time and buffer size). However, not every state variable contributes to self-adaptation, even if the variable can be modified dynamically. The proper set of feasible state variables should be deliberately selected during the development of a self-adaptive software system.

State variables are divided into the two categories of *controllable* and *non-controllable* variables. A non-controllable variable may represent a phenomenon that is not directly manageable (such as response time), or may represent a disturbance (such as user traffic). Sometimes, an attribute may not be measurable directly, such as performance. Therefore, proxy attributes like response time and throughput are required for measuring performance.

Capturing perceptive state variables in software models increase our insight about the feasibility of observing and controlling these variables. For example, in a multi-threaded application, two or more threads might share a single state variable. Controlling the state of such an application by altering the value of this kind of state variable is not straightforward, as we may introduce inconsistencies, race conditions, or even deadlocks. Thus, depicting state variables can help us to identify those variables that can be feasibly and safely observed and controlled for the purpose of adaptation.

3.4.2 Effecting Styles

Adaptation is based on change and variability, and the adaptation manager needs effectors to apply changes to the adaptable software. Our approach for analyzing effecting styles is based on the control-flow analysis of object-oriented applications, as shown in Table 3.2.

Table 3.2: List of Main Effecting Styles

Changes	**Name**	**Description**
Flow	Replace Block	Replace a code block with its alternative.
	Select Block	Alternate control flow between a code block and its alternative blocks.
State	Change State	Set software state by either setting state variables directly, or through state variable accessor methods, or other state definers.

This classification has two distinct sets of effecting styles for *what* we are effecting: i) those that alter an application's behaviour via structural changes in the application's control flow (*Flow*), and ii) those that preserve the application's flow, while modifying the application's operating states in order to attain a plausible behaviour (*State*). This classification matches two general approaches for implementing software adaptations: *parameter adaptation* for behavioural changes, and *compositional adaptation* for structural changes [125]. Concerning *when* the effects take place, we distinguish two distinct sets of effecting styles for runtime (dynamic) and pre-runtime (static) adaptation [125].

Replace Block

This effecting style replaces a block of control flow with another block at runtime. To implement this mechanism, we should have well-defined interfaces at cutoff points. The mechanism also demands support for dynamic binding, such as the program's ability to bind and unbind components (or modules) at runtime. The granularity level of this mechanism depends on the programming-language features and the application's container. This effecting styles can be realized by meta-class and meta-object reflective models at various abstraction levels [35]. *Replace Block* can be supported by language-based dynamic compositional techniques (e.g., *reflection* in Java, and *function pointers* in C/C++), or middleware-based approaches (e.g., dynamic aspect weaving, and dynamic EJB deployment in JavaEE) [125].

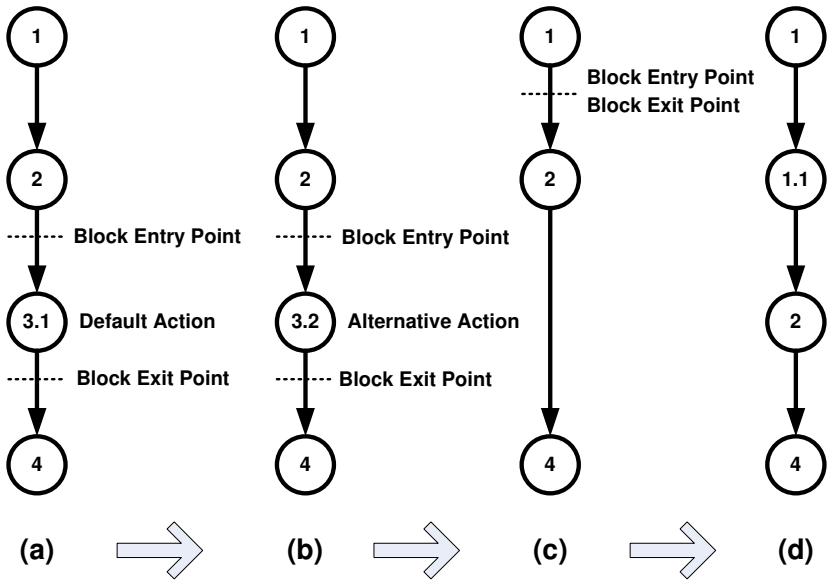

Figure 3.8: The Replace Flow Mechanism

Figure 3.8 illustrates three possible usages of the *replace block*. Figure 3.8.a represents a flow graph of an application. With the support of dynamic module binding, the adaptation manager should be able to replace node 3.1 with 3.2 as shown in Figure 3.8.b. Node 3.2 has to satisfy the required contracts (pre-conditions and post-conditions) at the starting (and ending) cutoff points of module 3.1.

Null actions are default or alternative actions that do nothing during program execution. We can make use of null actions to add or remove real actions. Figure 3.8.c shows the replacement of node 3.2 by a null action. Furthermore, we can also replace a null action (at a single cutoff point) by a new flow as demonstrated in Figure 3.8.d, by adding node 1.1 between the node 1 and 2 in Figure 3.8.c.

Select Block

This style redirects the program's *default flow* to an *alternative flow* at a *turning point*. A turning point is a branch in the control flow with a well-defined *decision criteria*. The decision criterion is a logical expression that navigates the runtime flow to an appropriate path between the default and alternative actions.

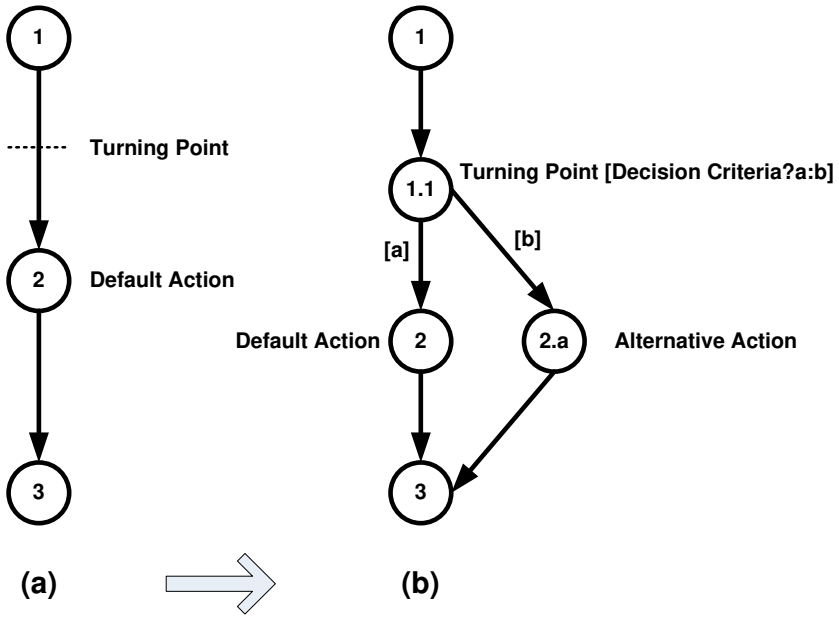

Figure 3.9: The Switch Flow Mechanism

Figure 5.4 uses a simple example to illustrate the *switch flow* style. Figure 5.4.a represents the flow graph of a non-adaptable software application with a detected missing turning point at flow $< 1, 2 >$, and a single default action of node 2. In Figure 5.4.b, the same graph is augmented by an alternative action at the turning point, at node 1.1. The decision criteria of node 1.1 can be set to select either the default action at node 2, or the new alternative action, at node 2.a. If the decision criteria is set in such a way that makes the alternative action unreachable, the adaptable software will behave like the original software. However, the adaptation manager can control the application's flow by adjusting the decision criteria.

Change State

Each state variable has zero to many *definer* and *user* statements. *Definers* aim to change an application's behaviour by resetting the application's state. On the other hand, the *change-state* style may benefit from *user* statements for observing the application's state, and checking consistency before altering the state variables.

The change state style does not modify the application's control flow and it can be added to software without prior flow analysis. However, the style should be used with

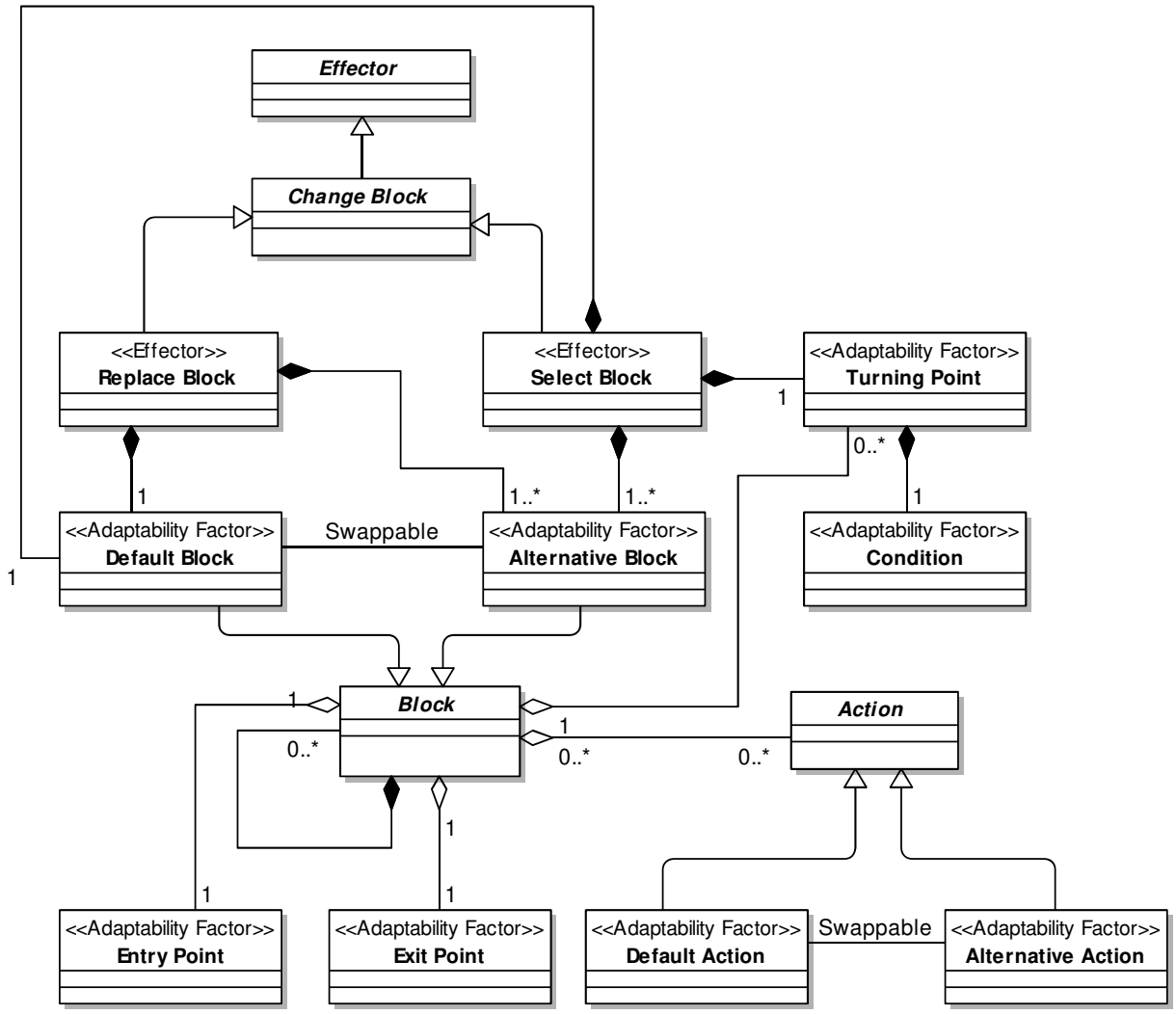

Figure 3.10: The MetaModel for Software Adaptability - Change Flow

care in case that state variables are scattered throughout the application. Multi-state *definers* and *users* may lead to inconsist application behaviours, or cause the system to become uncontrollable. An example of a runtime adaptation approach for altering state variables uses *setter* and *getter* methods is *tuning parameters* [47]. These methods can be exposed through an interface for easy access by external controllers. This approach is straightforward due to the minimal required changes in the application.

Structural effecting styles (i.e., *replace block* and *select block*) may also benefit from *change state* to include external effecting actions. We can expose the decision criteria of the *switch flow* technique as a state variable, and control this variable via its *definer*. Additionally, *replace flow* and its enabling techniques, such as structural reflection, can utilize structural effecting styles to change the value of the meta-layer's state variables.

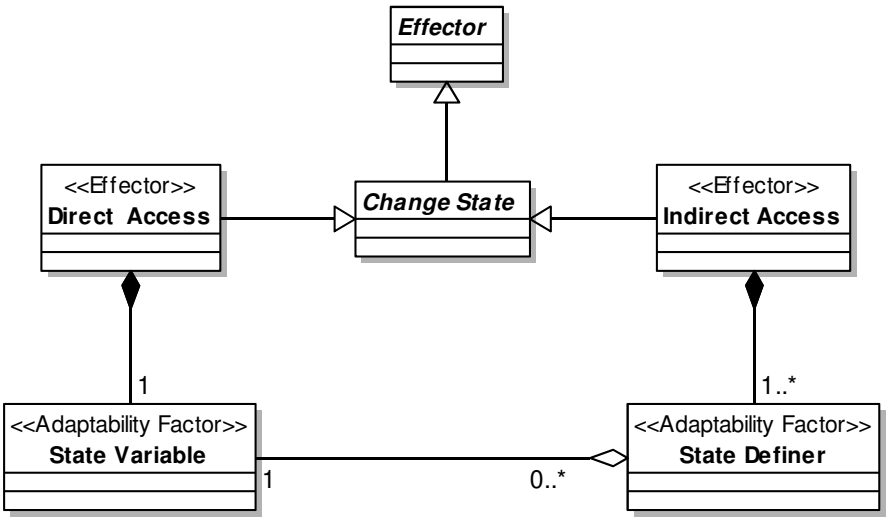

Figure 3.11: The MetaModel for Software Adaptability - Change State

Change state can also set the starting state of a module by adjusting its configuration variables. Configuration will be carried out only once at startup time. Note that the modules can be deployed and started dynamically by utilizing other effecting styles, which requires appropriate state initialization.

3.4.3 Sensing Styles

In deterministic software, we can understand an application's behaviour by observing its state [133, 47]. However, this procedure is not always feasible, as most state variables are implicit and hidden from the external viewers of the system.

For example, Garlan *et al.* propose a sensing style for architecture-based self-repair systems [68]. The focus of this work is on monitoring and detecting run-time violations from architecture styles. Moreover, Shaw *et al.* present a feedback control architecture that can be mapped to this sensing style [133]. In this architecture, the sensors are only used to observe the current state of the executing system.

The common property of all these approaches is using a style to observe states of the software at runtime. However, an adaptation manager might control adaptable software by mining its historical data (past sates), or predicting its future states. Unfortunately, these styles (or any other potential styles) are not mature enough to be classified as a separate sensing style in our metamodel. Hence, we classify all of them as a unified sensing style.

Monitor State

Observing software for changes in itself (or in its domain) is achievable through monitoring states or monitoring the invocation of the actions that can possibly change states (i.e., state *definers*). Analogously to *change state*, in *monitor state*, not every state variable is observable in the application's behaviour, and state variables should be carefully selected for observation, so as to prevent unwanted overhead on system performance.

Entities that support *monitor state* are the state *definer* and *user* methods (e.g., *setter* and *getter*) of the state variables. *Monitor State* can be used to implement diverse observing techniques, as listed in [165]. For example, logging is a sensing style that can be implemented as a set of state *user* methods that expose the system's time and state variables selected from a specific location of the system.

3.4.4 Adaptability Factors

Adaptability factors are abstract composing elements of effecting styles and sensing styles. We investigate each primitive style for its set of adaptability factors. A complete list of the captured adaptability factors is given in Table 3.3.

In adaptable software, there is no difference between the *default action* and *alternative action* at runtime. However, we consider them as distinct adaptability factors in Table 3.3.

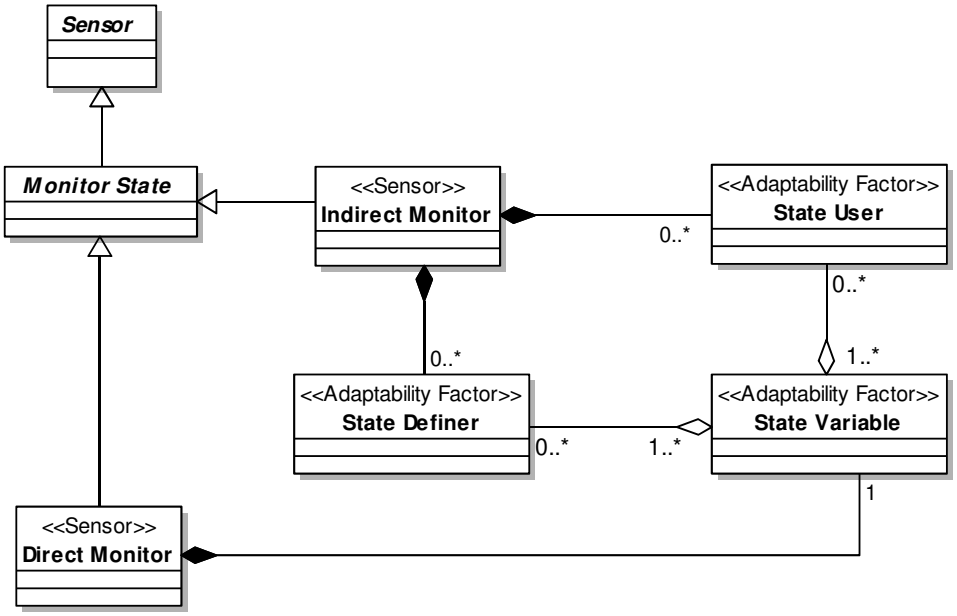

Figure 3.12: The MetaModel for Software Adaptability - Monitor State

This separation, which is also reflected in our metamodel, holds information on software commonality and variability [43]. This meta-information can be used by adaptation managers for commonality and variability (SCV) analysis, and to increase the understanding of the control flows by setting apart those actions that are added by the evolution processes for achieving adaptability.

3.4.5 From Conceptual to Concrete Adaptability Models

The proposed metamodel for software adaptability is comprehensive in terms of its capability to cover all types of effecting styles and sensing styles. This metamodel can be used to design and implement software adaptability. One way to achieve this goal is to express adaptability specifications by a modeling language. Such a domain specific language for modeling software adaptability should include new modeling elements for describing sensing and effecting styles as a set of adaptability factors.

An adaptability model can be specified as an extension of existing software models, as long as we can extend the existing metamodels with concepts for expressing the properties of software adaptability. The two most common abstraction levels are at: (i) the *code-*

Table 3.3: List of Identified Adaptability Factors

Adaptability Factor	**Description**
Default Block	A code block which implements the non-adaptable behaviour
Alternative Block	A substitute code block that implements adaptive behaviours
Default Action	An action of a non-adaptable application
Alternative Action	A substitute action, taken to adapt the program
Turning Point	A branching point in a program flow for switching between default and alternative actions
Condition	A logical expression to be evaluated at a *Turning Point* to choose appropriate actions
Entry Point	An ending point of an activity block that can be replaced with another activity
Exit Point	A starting point of an activity block that can be replaced with another activity.
State Variable	The current state of an application
State Definer	An operation that defines a value for a state variable.
State User	An operation that uses a value for a state variable.

level in which adaptability is expressed in terms of either direct annotators of the source code [2], extensor of the programming language, or domain specific languages; and (ii) the *design-level*, in which adaptability is expressed by existing behavioral and structural design models (e.g., UML activity diagrams and class diagrams).

Each abstraction level has advantages and disadvantages. The code-level does not require design-model extraction; it also keeps the adaptability model within the source code, which reduces maintenance costs and the chance of introducing inconsistencies as software evolves. The design-level approach specifies adaptability at a higher abstraction level, which is beneficial for large and complex systems, because unwanted source-code complexities are avoided, and because both platform-specific and platform-independent specifications of adaptability can be expressed. However, the design-level approach requires an updated version of the models to be extended and tagged.

Adaptability entities are not explicitly expressed in current software models; to express such entities modeling languages can be extended to include sensing and effecting styles and their composing adaptability factors. Because sensing and effecting styles deal with both behavioural and structural aspects of software, the extended languages are also required

to support the modeling of both aspects. For example, UML and its metamodel can be extended and used as a modeling notation for expressing software adaptability factors via profiling. UML profiles allow the specification of new stereotypes, tagged values, and constraints [140]. They can be applied to UML-diagram elements for expressing adaptability concerns.

3.5 Summary

As discussed in this chapter, the problem domain of self-adaptive software is divided into the two main problems of adaptation and adaptability. Given a set of adaptation requirements, specifications need to be defined for the adaptable software (i.e., adaptability specifications) and the adaptation manager (i.e., adaptation specifications). The conceptual model and the metamodels described, assist requirements engineers in understanding the SAS domain, and in specifying the adaptation manager and adaptable software based on the shared phenomena of their associated domains.

However, the proposed conceptual model is not an architecture model for SAS. For example, compared to the IBM Autonomic Computing reference architecture [98], this model is more generic, in the sense that it addresses the problem space with regards to architectures as an abstraction of the solution space. In the Autonomic Computing model, the interfaces between the adaptation manager and the application domain do not exist. In other words, its adaptation manager only has shared phenomena with the adaptable software, but does not share any phenomena with the domain. Hence, for open-loop control, the sensor and effector interfaces should be exclusively specified based on the shared phenomena between the adaptation manager and the adaptable software.

In the next chapter, we will propose a model-centric self-adaptation approach, which aims at reducing the complexity of engineering and maintaining control-based SAS, by observing and controlling partial and customized adaptability models of the software at runtime instead of directly managing adaptable software. The approach is realized in a generic and systematic manner, as a framework called GRAF. Using GRAF, what we reify from the adaptable software as state variables is observable via querying the runtime model, and what we reflect is controllable via transforming the runtime model.

Chapter 4

A Model-Centric Self-Adaptation Approach

"...with proper design, the features come cheaply. This approach is arduous, but continues to succeed."
–Dennis Ritchie

The complexity of SAS manifests itself in all composing elements of the system: the application domain, adaptable software, adaptation manager, and communication interfaces. Different solutions for enabling adaptivity in software systems can affect the level of complexity of each of these entities in different ways. For example, external approaches to SAS naturally increase the communication complexity of the system because of the communication channels between the adaptation manager and the adaptable software. In contrast, following internal approaches can inversely affect the quality of the final system due to the high coupling between the adaptation logic and application's business logic. For many real-world scenarios, this undesired *accidental complexity* [23] detracts from the advantages of moving towards self-adaptivity.

The use of simplified and more abstract models is a common general strategy for dealing with complexity in software engineering by finding generic solutions and decisions [120]. Models specify abstractions of real-world phenomena in such a way that the entities and relationships among them are simplified; the abstraction level can vary and depends on the application domain. Such models are used for a variety of purposes; for example, in software construction, abstract models serve as the basis for transformations resulting in the generation of (executable) source code. From this perspective, models are often used during the software development cycle and prior to runtime.

In the model-centric approach to runtime adaptivity, models of software need to be generated, manipulated, and managed at runtime [69, 131]. These models, known as *runtime models*, serve as meta-layers for software in a reflective architecture. Moreover, an *adaptation manager* controls the adaptable software by manipulating its runtime model instead of directly operating on the adaptable software. This reflective architecture model is sketched in the context diagram illustrated in Figure 4.1.

To achieve adaptivity, the state of the adaptable software is propagated to its runtime model first (*reification*). These changes are then observed by the adaptation manager (*sensing*). Subsequently, the adaptation manager plans and selects adjusting actions (*controlling*) and adapts the runtime model accordingly (*effecting*). Finally, changes made to the runtime models are propagated back into the adaptable software (*reflection*). The structure and behavior of software that is built around this *model-centric architecture* can be changed by modifying only the models. In addition, model transformations can support the implementation of adaptivity by modifying models at runtime.

The runtime model in a model-centric approach represents a *connection domain* [89] between the *adaptable software* and its *adaptation manager*, with respect to the reference architecture of an SAS [98]: the run-time model offers a specific view of the software's core functionality and contains (i) *behavior descriptions* for adaptation purposes, as well as (ii) parts of the adaptable software's *state*.

According to M. Jackson, there are three approaches for dealing with the runtime model as a connection domain [89]: (i) ignoring the runtime model, and dealing directly with the adaptable software, (ii) treating the runtime model as if it was itself the domain of interest, and (iii) recognizing that the runtime model and the domains it is representing (the adaptable software and its application domain) are both important.

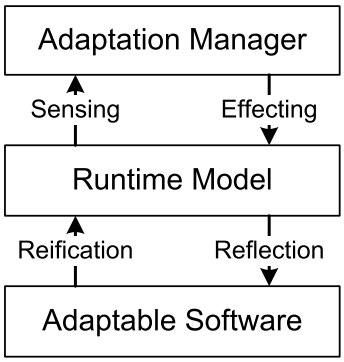

Figure 4.1: Context Diagram of a Model-Centric SAS [50]

The first approach represents the classical view of self-adaptive software, where the adaptable software is directly connected to and controlled by its adaptation manager, as in Equation 3.3. However, in the second approach, the adaptation manager interacts with and controls the runtime model directly. This is a common practice in recent approaches for creating self-adaptive software (e.g., [132, 39]). This approach highlights the role of runtime model (RM) and enables us to rephrase Equation 3.3 as:

$$RM, S_{Adaptation} \vdash R_{Adaptation} \tag{4.1}$$

Equation 4.1 is valid at runtime, and requires that the runtime model is constructed and maintained independently. specifically, in the case of migrating existing software towards runtime adaptivity, the second approach is not sufficient; in this case, we consider the third approach: the original software cannot be ignored, because establishing proper relationships between the adaptable software and its runtime model is a significant sub-problem. One solution to this sub-problem is to explicitly maintain the relationship between the adaptable software and its runtime model. In this approach, a software system is considered adaptable if there exists a runtime model of the system that is observable and controllable by an adaptation manager. When migrating existing software towards adaptivity, the *adaptable software* is derived from existing software that contains the business logic. In an alternative scenario where self-adaptive software is to be constructed from scratch, the adaptable software is replaced by a subsystem that merely provides basic functionality to be used as building blocks during construction.

Figure 4.2 illustrates how the runtime model fits between the adaptable software and its adaptation manager, and sketches the relationship among the adaptation specification, the adaptability specification, and the main components of a model-centric SAS.

Here, the shared phenomena between adaptable software and adaptation manager are the states and the actions, and the elements of the runtime model are abstractions meaningful in software that their roles and interactions reflect the adaptation workflow. Domain attributes, which are to be observed by adaptation manager, represent the state of the software or its application domain. Moreover, adaptation actions in the model-centric adaptation are applied to a runtime model instead of directly to the adaptable software; this means that the action specifications in the adaptation manager remain the same, but the executing specifications are changed.

Although model-centric approaches to runtime adaptation seem promising; creating, managing, verifying, reflecting, and keeping the consistency of runtime models adds additional complexity and overhead to the system. This drawback can put the usability and

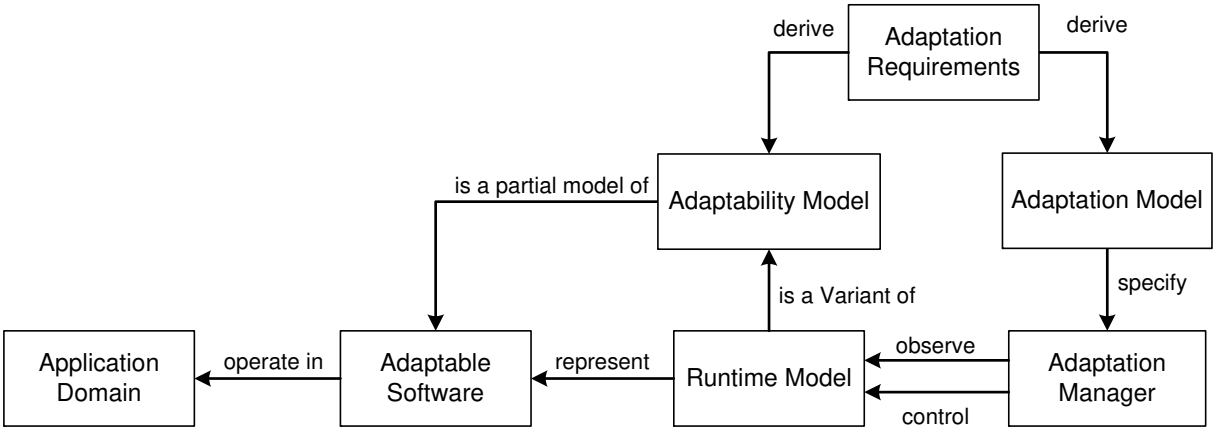

Figure 4.2: Relationship Among the Abstract and Concrete Entities of a Model-Centric SAS

cost-benefit ratio of model-centric approaches to SAS under question. In order to tackle this problem, adaptation frameworks that are specially tailored to contain and manage models at runtime can be used to reduce the complexity of using models at runtime and enhance the efficiency of model-centric approaches.

Following model-centric approach to achieve runtime adaptivity, we designed and developed the *Graph-based Runtime Adaptation Framework* (GRAF). GRAF is a model-centric adaptation framework that explicitly separates the adaptable software from its runtime model to isolate adaptivity concerns from the rest of the business logic. The framework supports the separation of supervision and control from the application's core functionality by realizing the adaptation manager externally, instead of mixing it with the adaptable software itself.

We present the structural and behavioural architecture model of GRAF. The design of GRAF's architecture is based on the reference architecture for self-adaptive software, which consists of a *controller* and the *controlled plant*, as discussed in Chapter 3. Following this architecture, the overall structure of a GRAF-based SAS is separated into two main subsystems: (i) the *adaptation framework* and (ii) the *adaptable software*. as shown in Figure 4.3 high-level component model. Both subsystems communicate with each other via predefined interfaces. In this approach, the adaptation manager's input and output data are respectively encoded as model queries and transformations on the runtime model.

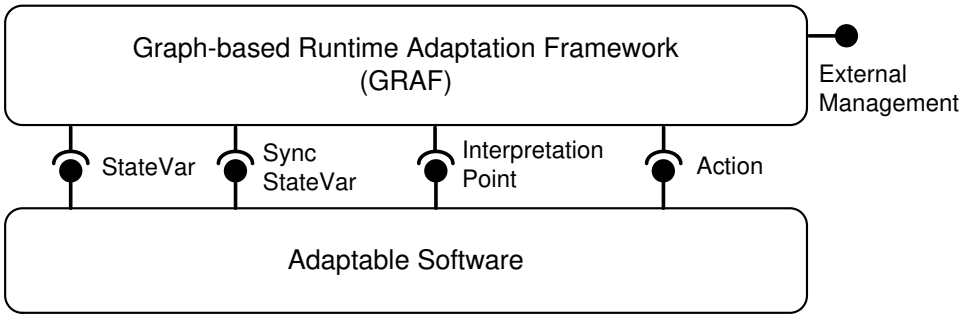

Figure 4.3: High-Level Component Model of GRAF-Based SAS

4.1 Design Considerations

Before describing the framework's architecture and runtime behaviour in detail, we first introduce the main design considerations of GRAF.

4.1.1 Model Interpretation for Change Reflection

Given that the meta-layer is available as a model, the process of *reflecting*, i.e., injecting changes from the meta-layer into the base layer, can be done in several ways and using different techniques. For instance, software components can be composed at runtime [39, 193] or functionality can be adjusted using dynamic aspect weaving [74].

GRAF combines the concepts of having a reflective architecture with the *interpretation* of models at runtime, thereby following an approach to building generic software that is based on composing components that depend on models to describe the variable parts of the system. Such explicit models are then *interpreted* at runtime by a generic and stable core.

The meta-case tool *KOGGE* is an example of software constructed around this mechanism [59]. Motivated by its flexibility, we decided to follow a model interpretation approach for GRAF as well. Hence, the main way of achieving adaptivity in this research is by redirecting the adaptable software's control flow to a model interpreter component at points where the need for adaptivity is expected.

Besides achieving adaptation by adjusting program variable values, the main way of achieving adaptivity in this research is by redirecting the adaptable software's control flow to a model interpreter component at points where the need for adaptivity is expected.

When such an *interpretation point* in the control flow is reached during the execution of the adaptable software, the model interpreter executes an associated behaviour, as described in the runtime model. Therefore, transforming behaviour descriptions that are stored in the runtime model results in adapted program behaviour.

4.1.2 Model Verification

In a model-centric SAS system, the runtime model is transformed from a source to a target model at runtime. The changes between these two models specify the adjustments to be made to the adaptable software. Prior to reflecting them, it is important to verify the current runtime model after it is changed to ensure that certain properties hold.

Model verification at runtime is crucial, because a broken runtime model will likely result in faulty and undesired behaviour of the dependent adaptable software, or will eventually lead to a crash of the whole SASS. This concern is addressed in three different ways:

First, the runtime model is restricted by a *schema* (meta-model), which is itself a model, that formally describes the allowed modeling elements and their relationships. A runtime model can thus be checked in terms of conformance to its schema - that is, it is possible to verify its correctness with respect to abstract syntax. More complex context-sensitive constraints can be additionally defined by using a suitable constraint language. Similarly, a runtime model can be checked against these restrictions.

Second, the adapting transformations carry *pre- and postconditions* to be checked before and after execution. This supports the formal verification of possible and impossible runtime model states for a given set of transformations. This is important, because the adaptable software may not be changed under certain conditions and the set of potentially possible adaptation steps must be limited to the desired ones. Moreover, the effects and interactions among adaptation steps become transparent and may be better understood, especially in the context of maintaining an SAS.

Finally, in the case that the runtime model is transformed and, as a result, becomes invalid with respect to its schema or additional constraints and invariants, it is essential to allow *rolling back* any changes that led to this new model state. Obviously, the adaptable software's functionality must not be negatively affected by this recovery process.

4.1.3 Model Synchronization

In model-centric SAS, the runtime model is a shared resource between the adaptation manager and adaptable software, and the parallel execution of the adaptation manager and the adaptable software may lead to inconsistent runtime models. Resolving consistency issues in a model-centric SAS increases the accidental complexity. On the other hand, complete serialization of the adaptation loop with the application control flow can negatively affect the performance of the model-centric approach to runtime adaptation. Hence, one of the main challenges for supporting adaptation frameworks is to manage runtime models and their consistency with only minor overhead in memory and execution performance.

4.1.4 Separation of Concerns

Separating application logic from adaptation logic is an important design concern for self-adaptive software systems. This includes: (i) adaptation knowledge represented by attribute, action, and goal spaces, and (ii) adaptation mechanism implementation and other helper modules such as a policy engine which interpret and execute adaptation logic.

Similar to separation of concerns as a general principle in software design, this issue has tremendous impacts on the maintainability, extensibility, and testability of an adaptation manager. From another point of view, due to the fact that SAS mainly emphasizes quality requirements, the specification and control of these requirements have been separated from the functional parts of the application.

4.1.5 Low Coupling to Adaptation Framework

Assume that some software already exists and shall be evolved towards an adaptive system. In the case that no adaptation was performed at runtime, the resulting SAS has to perform the same functionality as the non-migrated legacy system. An obvious strategy towards this goal is to make as little changes as possible to the legacy system. Furthermore, the changed source code needs to be clearly marked. All changes to existing code must preserve the original behaviour. To further reduce the risk of undesired changes as a side effect of the migration to SAS, our goal was to limit the coupling between the adaptable software and the adaptation framework.

We decided to automate the needed binding between the adaptable software and GRAF for reifying and reflecting changes to and from the runtime model by using code injection mechanisms of aspect-oriented programming. Changes in the application are marked using

predefined annotations. The connection to GRAF is then realized by aspects, i.e., code that is also provided by the framework and that is woven into the compiled source code of the adaptable software. The spots for injection are pre-determined using generic *pointcut* expressions that are based on the annotations. In addition, an initial runtime model is also generated automatically from these annotations.

4.1.6 Extensibility and Reusability

In the design and development of an adaptation manager, modularity and reusability are important. Modularity eases the development of the adaptation processes, and also enables the adaptation manager to use external pre-built modules like policy engines for implementing processes. On the other hand, reusability enables us to use patterns for designing the adaptation model/mechanism and common modules for implementing them.

To address these two concerns, GRAF represents several composition patterns for designing the adaptation model and corresponding mechanisms.

4.2 Adaptable Software

The main property of the adaptable software is that it can interact with GRAF via predefined interfaces. There are four different interfaces, as illustrated in Figure 4.4: (i) `StateVar`, (ii) `SyncStateVar`, (iii) `InterpretationPoint`, and (iv) `Action`. The main task of these interfaces is to allow access to data and to expose building blocks for describing behaviour within the target domain of the adaptable software's business logic. The functionality of each interface from a high-level perspective is as follows:

- The `StateVar` interface is responsible for exposing *state variables* to GRAF or any other external adaptation manager. A state variable is a data field inside the adaptable software that contains information to be observed by GRAF. In essence, this interface exposes data that represents a part of the adaptable software's *state*. GRAF observes this state and may react to its changes.

- The `SyncStateVar` interface is an extension of `StateVar`, which in addition to exposing state information, allows the modification of the state. This enables GRAF to change the state of the adaptable software by adjusting its exposed state variables. The name of this interface is derived from the fact that the value of state variables

within the adaptable software is *synchronized* with, and in fact overridden by, data that is computed by GRAF and stored in the runtime model.

- The `InterpretationPoint` interface exposes points in the control flow of the adaptable software, such as the entry points of method calls. These are points in the adaptable software at which behavioural variability is needed. By exposing such interpretation points, a predefined section of the adaptable software's runtime behaviour can be replaced by some alternative behaviour that is constructed by GRAF. This alternative behaviour is executed by interpretation, which also inspires the name of this interface.

- The `Action` interface exposes a set of elementary building blocks to be used for constructing behaviour descriptions in the runtime model. As soon as the control flow of the adaptable software reaches a point that was exposed to GRAF via the `InterpretationPoint` interface, an alternative behaviour may be available. This behaviour is composed by GRAF using actions. During the interpretation of the behaviour, the involved actions are called. In the context of modernization, actions that exist in the code are called *default actions*, and actions added to achieve adaptivity are called *alternative actions*.

Existing program elements of adaptable software, such as classes and methods or data fields, are considered to be the *original elements*. In order to build an adaptable software, original elements that hold phenomena to be shared with the adaptation manager need to be modified. If no such element represents the required shared phenomena in the software, new elements are added. These modified or added elements that are needed for achieving adaptability are called *adaptable elements*.

4.3 Adaptation Framework

The core components of GRAF are organized into four layers: (i) *adaptation middleware*, (ii) *runtime model*, (iii) *adaptation management*, and (iv) *management extension*.

The adaptation framework also provides an external management interface, which supports externally managing subsystems.

Throughout this section, we introduce each layer and its constituent components in terms of their functionality, tasks, and responsibilities. This section focuses on giving a *structural view* of GRAF, although some dynamic aspects are discussed as well.

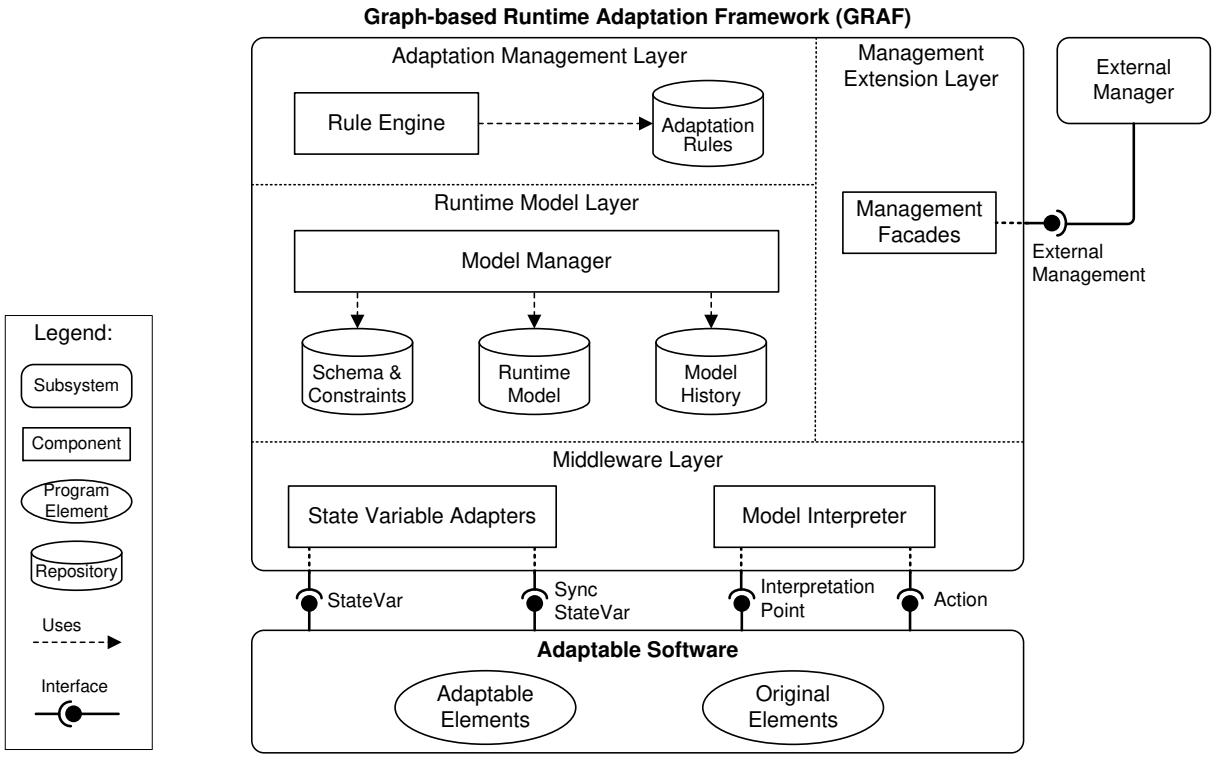

Figure 4.4: The Model-Centric Target Architecture of GRAF-Based SAS [50]

4.3.1 Adaptation Middleware Layer

The *adaptation middleware* is responsible for simplifying the connection of adaptable software to GRAF. Hence, the components in this layer are connected to the provided interfaces of the adaptable software.

State Variable Adapters

The framework offers a set of type-specific *state variable adapters* for passing data from state variables in the adaptable software to GRAF (`StateVar` interface). Similarly, whenever GRAF computes new values for a state variable, this data must be made available to the adaptable software (`SyncStateVar` interface). For the purpose of adjusting the adaptable software, the state variable adapters enable *parameter adaptation*.

Model Interpreter

For adapting behaviour and not just pure data, GRAF enables providing an alternative behaviour description to replace parts of the adaptable software's behaviour at predefined points in its control flow. The *model interpreter* component is responsible for enabling this functionality. The model interpreter is connected to the `InterpretationPoint` interface and is invoked at runtime, whenever a method that can be adapted is entered. If there is an alternative behaviour proposed by the framework, the model interpreter executes this description, which is a composition of elementary actions provided via the `Action` interface. For the purpose of adjusting the adaptable software, the model interpreter enables *compositional adaptation*.

4.3.2 Runtime Model Layer

At the heart of GRAF is the *runtime model layer*. It is essentially a meta-layer that enables runtime adaptivity.

Model Manager

In order to handle the runtime meta-layer in a uniform way, the *model manager* is responsible for all interactions with the three repositories (*Schema and Constraints*, *Runtime Model*, and *Model History*) located at the runtime model layer. The repository access operations offered by the *model manager* can be categorized into *query* and *transformation* operations. Both the adaptation middleware and the adaptation management layers contain components that interact with the runtime model layer via the model manager.

Runtime Model

The *runtime model* is a repository, or more specifically a model, that represents: (i) parts of the adaptable software's *state*, and (ii) a collection of *behaviour descriptions*.

The language for expressing runtime models can vary. In the current implementation of GRAF, behaviour is modeled using a subset of UML activity diagrams, where each `Action` element corresponds to an action that must be provided by the adaptable software.

The software state is modeled by counterparts for each state variable in the adaptable software. This part of the runtime model is glued to the adaptable software via the state

variable adapters described earlier. The behavioural part of the runtime model is used by the model interpreter. Each behaviour description is bound to an interpretation point and can then be executed.

The runtime model changes over time, first, due to values it receives from the state variable adapters, and second, due to its transformation by the adaptation managers. Generally speaking, the adaptable software is indirectly adapted. Its adaptivity-specific meta-layer, the runtime model, gets transformed first, and due to the causal connection that is established via the adaptation middleware, the adaptable software is finally adjusted.

A default runtime model can be generated for an existing adaptable software. GRAF has a *model generator* tool that reads the compiled source code of the adaptable software and generates a default runtime model for it. The model contains counterparts for each state variable, as well as behaviour descriptions for interpretation points.

Schema and Constraints

A *runtime model schema* describes the allowed model elements for a runtime model in GRAF. An excerpt of this schema is shown in Figure 4.5 and is expressed using the concrete syntax of UML class diagrams. The schema consists of elements that represent state variables as well as a subset of UML activity diagrams for modeling behaviour. By default, the runtime model schema defines model elements such as `InitialNode`, `FinalNode`, `OpaqueAction`, and `Flow`, which are very similar to those of the *UML Superstructure* defined by the OMG [78]. The full runtime model schema contains a nearly complete model of the abstract syntax for UML activity diagrams and covers state variables for the most frequently used primitive data types.

The schema defines the most fundamental *constraints* on any possible runtime model. More specifically, a syntactically valid runtime model *conforms* to its schema. It is possible to express constraints at (i) the *schema-level* and (ii) the *model-level*. The first constraint type is embedded into the schema and assures certain properties; for example, "the `declaringElementId` field of an `Action` element may not be empty". Such constraints are independent from the domain of the adaptable software.

The second constraint type allow users to specify domain-specific constraints; for example, "only specific values are valid for a given state variable", or "a sequence of `OpaqueAction` elements with certain names may not exist". These constraints are enforced by either adaptation requirements or the adaptable software. GRAF provides infrastructures to modify (add/remove/change) these constraints at runtime as needed.

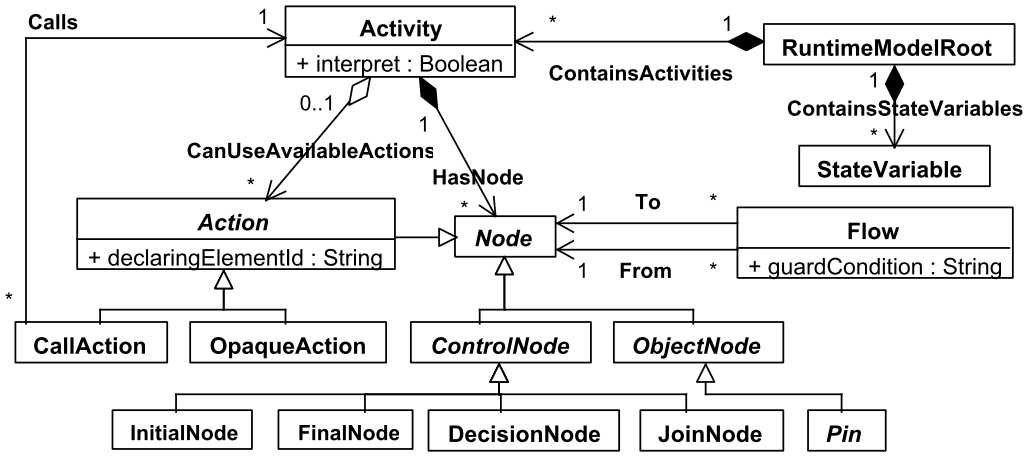

Figure 4.5: Excerpt of the Runtime Model Schema (MetaModel)

Model History

Given that the runtime model is transformed over time to adjust the adaptable software, there can be situations where past state(s) can be useful for adaptation management. The *model history* supports exactly that by offering versions of the runtime model that capture how the model has changed over time. The model history can be queried similarly to the runtime model itself. Sample use cases for the model history includes:

- Identification of alternating sequences of runtime model states due to a transformation.
- Identification of patterns in state variable values for prediction.
- Detection of invalid state variable values according to constraints.

Note that not only an autonomic adaptation manager can use the model history. Human administrators may choose to debug the self-adaptive software as well by using specialized views on the runtime model, supported by data mining approaches that operate on the model's history data; using this approach, observed states of the runtime model can be used to derive and predict potential future states.

4.3.3 Adaptation Management Layer

Up to now we described most of GRAF's components but did not state where the actual adaptation is performed. The adaptation management layer represents the adaptation manager shown in Figure 4.1 and contains a rule engine and a repository of adaptation rules. This layer is extensible and is designed to support any adaptation managers that implement the provided interfaces.

Rule Engine

The *rule engine* is responsible for (i) *monitoring* the runtime model, (ii) *analyzing* its change, (iii) *planning* a sequence of adjustments, and finally (iv) *executing* the plan via the model manager. Hence, the rule engine performs the *MAPE-loop* [87].

In the context of GRAF, monitoring potentially involves listening to a set of change events triggered by the model manager. For each event, the rule engine querying the runtime model must analyze the situation and determine the current state of the system; at this point, the planner produces a plan, which is actually a compound transformation to be applied to the runtime model. Such a transformation can change any part of the runtime model, such as state variable values or behaviour descriptions.

Adaptation Rules

We assume that adaptation requirements can be used to derive a set of *adaptation rules*. Developing adaptation rules is an essential part of creating self-adaptive software based on GRAF. All adaptation rules are stored in a central repository that can be accessed by the rule engine as well as by human administrators. A single adaptation rule is therefore an *Event-Condition-Action* (ECA) rule with the following structure:

ON *Event* **IF** *Condition* **THEN** *Action*

- *Event:* Some change in the runtime model (e.g., a change to a state variable value).
- *Condition:* A boolean expression that is specified as a query on the runtime model. The condition specifies the current state of the system. The state of the system proposes a list of valid actions to be taken. The planner as a part of the rule engine will select the best action or a list of actions to be executed.

- *Action:* The actual change to be made to the runtime model in the form of a transformation. Each action can be enriched with pre- and post-conditions for verification and validation purposes.

As discussed in Subsection 3.3.4, adaptation managers are rational agents, and their behaviour can be expressed by describing the relationship between the states and actions needed to achieve adaptation goals. Although selecting the adaptation actions depends on the desired adaptation goals and properties, an adaptation manager, as a rational agent, observes the current state, and performs an action accordingly to make a transition to a new state. Therefore, the propositional logic nature of ECA rules is expressive enough, but not necessarily optimized, to describe the behaviour of adaptation managers.

4.3.4 Management Extension Layer

This layer is composed of a set of *facade* components and APIs. The *management extension* exposes an external management interface via a set of managed Java beans, called *MBeans*. They are bound to their corresponding components in each layer and are responsible for: (i) providing service access to GRAF, (ii) providing the external configuration of GRAF and its subsystems, (iii) serving as an observer for monitoring changes to the runtime model and providing notifications for its listeners using the events that the model manager raises when the runtime model changes, (iv) providing an interface to query and transform runtime models (via the model manager), which can also be used to add and update the application's domain (operating environment) state variables of the runtime models, and (v) providing access to the internal adaptation manager and adaptation rules, for the purpose of modifying adaptation rules at runtime.

The management extension layer also provides an API set that allows GRAF to act as a service, which can be activated and deactivated on the fly. In this case, the whole framework can be viewed as a wrapper service for adaptable software. This service provides dynamic access to the adaptable software's runtime model, allowing it to be observed and controlled from external resources.

External Manager

GRAF is able to accept *external managers* via the the external management interface. These components can extend the core functionalities of the framework or be used to observe and control a GRAF-based SAS externally. For instance, *external adaptation*

managers like *StarMX* [9] can connect to GRAF to query and transform the runtime model.

Moreover, manual interaction with an SAS system is feasible via well-designed user interfaces. Possible access methods range from text-based command line interfaces to remotely controlled and web-based applications. These *control panels* can be used for administration tasks, such as monitoring the operating state of the SAS and controlling its behaviour manually.

4.4 Runtime Behaviour

After introducing the overall structure and the features of GRAF, we describe the framework's behaviour at runtime, covering five main use cases. Subsequently, we give a description of these use cases in the form of activities, using the unified diagram in Figure 4.6. The first two activities are illustrated together though, because of their tight interaction.

Here, we deliberately ignore all possible alternative flows and exceptions that might occur during the adaptation. The main idea is to show the control flow of GRAF under *typical* conditions.

4.4.1 Reification and Sensing Use Cases

The *reification* involves the propagation of changed state variable values from the adaptable software to the runtime model. This results in the sensing use case that is notifying the rule engine.

Assuming application-level adaptation as discussed in [165], changes are reflected by the state of the adaptable software and thus can be sensed through its variables and fields.

The first use case starts with a change to the adaptable software and its data as the system communicates with its operating environment. If the changed variable is an instrumented Java field that is exposed via the `StateVar` or `SyncStateVar` interfaces, a state variable adapter of GRAF takes care of sending the updated value to the model manager, which decides whether the propagated value should be stored in the runtime model. Finally, the model manager notifies the rule engine whenever the model changes.

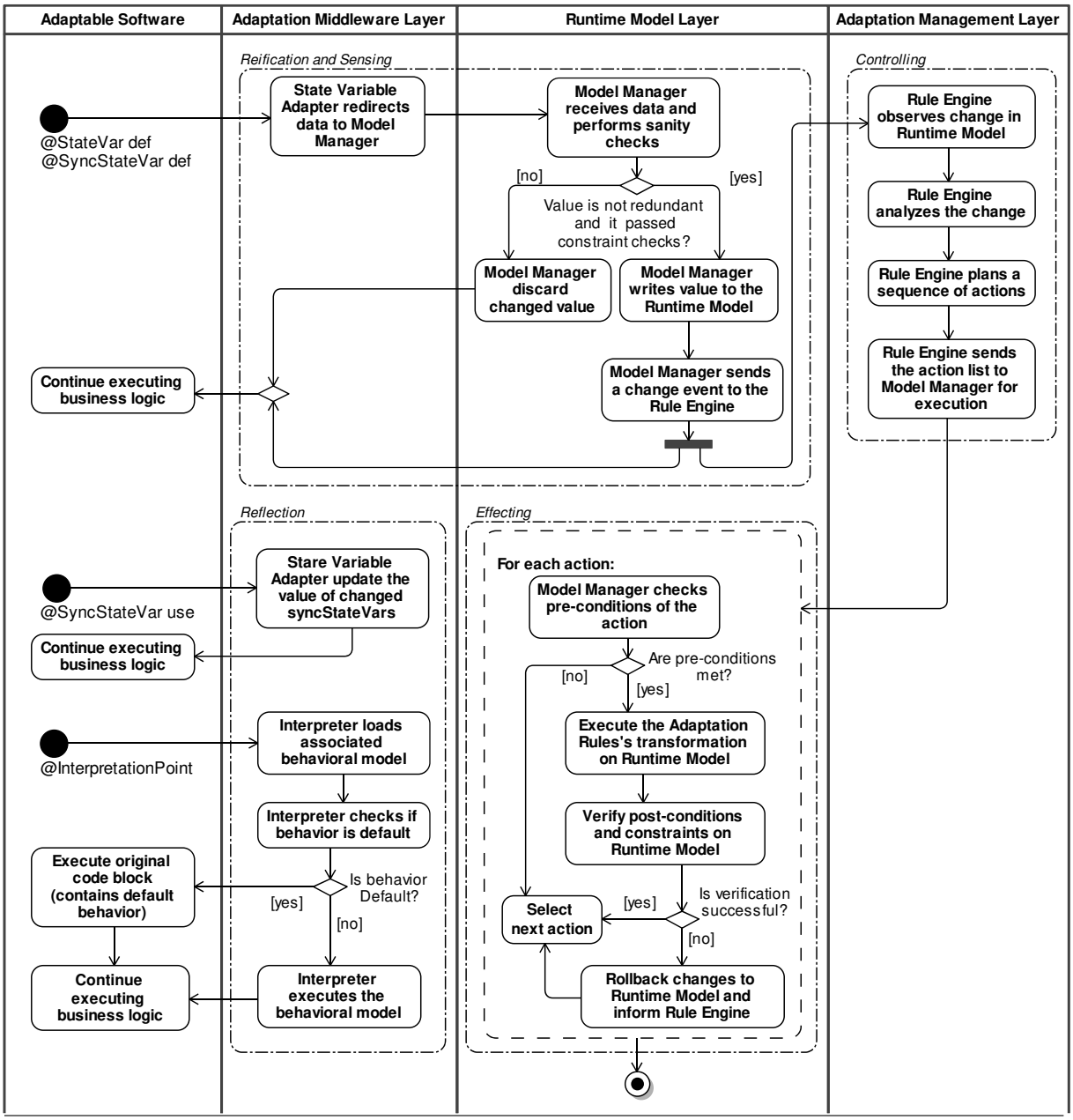

Figure 4.6: Main Control Flow of GRAF Runtime behaviour

4.4.2 Controlling Use Case

The IBM vision of autonomic computing introduces the *MAPE-loop* for external software control as a four step process: (i) *monitoring*, (ii) *analyzing*, (iii) *planning*, and (iv) *executing* [98].

The *controlling use case* of GRAF is essentially the *MAPE-loop*. The rule engine analyzes the changes, as soon as it receives a signal from the model manager. Then, it plans for the best possible set of actions (transformations) to be executed on the runtime model, considering the repository of available adaptation rules and based on its planning strategies. Optionally, the rule engine may use the runtime model and it's change history by querying them via the model manager for a more detailed evaluation, e.g., taking past adaptation steps into account. In the last step, the rule engine triggers the invocation of the action part of the adaptation rules (transformations) on the runtime model.

4.4.3 Effecting Use Case

The *effecting use case* of GRAF involves a transformation of the runtime model that is based on the adaptation rules selected by the rule engine in reaction to changes in the adaptable software.

First, the model manager checks the precondition of the adaptation rule on the runtime model. If it evaluates to false, the adaptation rule is ignored; otherwise, the adaptation rule's transformation is executed. Next, the model manager checks and verifies the runtime model against the schema and its constraints, as well as the adaptation rules's post-conditions. In the case of any conflicts and constraint violations upon the the executed transformation, the model manager rolls back all of the changes and notifies the rule engine to take another action. If the executed transformation is valid, the runtime model is successfully adapted to the new environment, based on the existing adaptation rules and given model constraints.

4.4.4 Reflection Use Case

The *reflection use case* of GRAF is responsible for an actual behaviour change in the managed adaptable software. Two different adaptation techniques are supported: (i) adaptation via parameter values, and (ii) adaptation via interpretation of a behavioural model.

Adaptation via State Variables

State variables can be setup so that their changes are reflected back to the adaptable software. For this mechanism to work, the corresponding variable of the adaptable application must be exposed to GRAF via the `SyncStateVar` interface. If the runtime model representation of such a state variable is changed by an adaptation rule, then the adaptable software becomes aware of the new value whenever the variable is *used* in the adaptable software.

Adaptation via Model Interpretation

GRAF supports the modeling of behaviour, currently using a subset of UML activity diagrams. The model interpreter executes an activity at specific points in the adaptable elements' control flow.

During the execution of some method of the adaptable software that is exposed to GRAF via the `InterpretationPoint` interface, the model interpreter is invoked and loads the associated behavioural model. If the model describes default behaviour, there is no need for adaptation and the interpreter returns this information so that the adaptable element executes the existing default code block in the adaptable software. Otherwise, the corresponding part of the behavioural model is changed to describe something other than the default behaviour, and the interpreter executes the behavioural model.

Each action used in the behavioural model is associated with methods in the adaptable software via the `Action` interface, and the interpreter can invoke these methods. The adaptable software keeps executing after returning from the potentially adapted method.

4.5 Realization

So far we have introduced the architecture for a GRAF-based SASS in terms of its static structure and its runtime behaviour. In this section, we describe a prototypical implementation and the technologies involved.

GRAF is implemented in Java and can be utilized by standard or enterprise Java applications for the purpose of runtime adaptation. In this section, we follow a bottom up approach and introduce the technologies used to create the functionality related to each subsystem and layer. However, we discuss only *one* possible way of realizing such an adaptation framework. A more detailed description on the presented techniques can be found in [49].

4.5.1 Runtime Modeling with the TGraph Approach

We implemented the runtime model layer using the *JGraLab* API for processing *TGraphs*; i.e., typed, attributed, ordered, and directed graphs. JGraLab can create Java source code based on meta-models (schemas). The generated classes can be used in custom applications to utilize TGraphs as a data structure. JGraLab can create and manipulate TGraphs at runtime.

The runtime model can be queried using the *Graph Repository Query Language* (GReQL) [57] and it can be transformed using the *Graph Repository Transformation Language* (GReTL) [84]. Furthermore, query and transform operations can also be realized using a lower level Java API that operates directly on the runtime model.

This API is auto-generated for the runtime model schema, which we describe via UML class diagrams using the IBM Rational Software Architect 7.5 [155]. An exported XML Metadata Interchange (XMI) [1] representation of the schema model is then processed by JGraLab tools to generate the mentioned Java API.

4.5.2 Model Interpretation and Reflection

The model interpreter walks along the paths of an activity diagram that describes the behaviour to be executed at an interpretation point. Hooks for model interpretation are injected into the bytecode of classes containing methods that are tagged as `@InterpretationPoint`.

Each `Action` element of the behaviour, modeled by the activity diagram, has a trace link to a method in the adaptable software that actually implements the functionality. The model interpreter uses information such as a fully qualified name to invoke methods via Java's built-in reflection mechanism.

To enable or disable the execution of the model interpreter, an activity model has a boolean `interpret` attribute that specifies, whether the behaviour description shall be interpreted. Initially, this flag is only set to `true` in cases when the described behaviour in the runtime model does not match the default one. We consider the default behaviour to be the same as already implemented by the adaptable software at a given interpretation point. This means that if the model manager states that the behavioural model described default behaviour, no changes to the adaptable software's control flow are made, but its

[1] The XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML) [139].

source code is executed as normal. Otherwise, the model interpreter actually changes the control flow by executing the associated part of the behavioural model. The original body is skipped, and the resulting value of the interpretation run is returned.

Hence, with respect to its implemented default behaviour, the adaptable software behaves similarly in terms of execution performance if we remove the additional check of the `interpret` flag. This is an essential requirement that stems from the focus on modernization towards runtime adaptivity.

Finally, the value of `interpret` can be modified during the preparation phase; for example, by the model generator, from adaptation rules, or even during a manual refinement phase of the runtime model. This makes it possible to fully rely on model interpretation only. This may be desired in scenarios, in which an SAS is being developed from scratch, where even default behaviour is modeled based on more generic methods.

4.5.3 Tagging Adaptable Elements with Java Annotations

As GRAF is mainly designed to facilitate evolving software system towards adaptivity, the main goal is to maintain the changes required to make software adaptable. The functional characteristics of the software must not change when it is decoupled from the framework. The adaptable software communicates with GRAF via a set of interfaces. In order to minimize the amount of source-code changes, instead of implementing real Java interfaces in each adaptable element, we decided to implement them in another way.

For each interface type, GRAF offers a corresponding Java annotation that has the same name. We make use of Java's common meta-annotations to set up each custom annotation type, such that it is applicable only to the Java element that it is intended for. For instance, there is a `@StateVar` annotation and it is only applicable to member variables (`@Target(ElementType.FIELD)`). Furthermore, the annotations are set up to be readable at runtime using Java's built-in reflection mechanisms (`@Retention(RetentionPolicy.RUNTIME)`).

In summary, the following annotations are offered by GRAF: (i) `@StateVar`, (ii) `@SyncStateVar`, (iii) `@InterpretationPoint`, and (iv) `@Action`. Adaptable (program) elements are marked in the source code using one of these annotations. This added meta-data is then used for binding the adaptable software to GRAF using *aspect-oriented programming* (AOP) techniques. Additionally, these annotations are used for generating an initial version of the runtime model.

@StateVar

Variables in adaptable software representing state variables as a part of the *Monitor State* sensing style are annotated as `@StateVar`. Changes in this data field will be propagated to GRAF's runtime model. Listing 4.1 shows an example where a integer value represents number of active users.

```
1 @StateVar
2 private int numberOfActiveUsers;
```

Listing 4.1: Use Of The @Statevar Annotation

@SyncStateVar

State variables that are part of the *Change State* Effecting Style are annotated as `@SyncStateVar`. The difference between these *synchronized* (sync) state variables, and state variables annotated by `@StateVar` is that if an adaptation manager computes a different value and sets it in the runtime model, this value will be written back to the adaptable software. Listing 4.2 shows an example where a `boolean` field is exposed for controlling by the adaptation manager. That way, code in the adaptable software can be written to depend on externally tuned data.

```
1 @SyncStateVar
2 private boolean blockNewUsers;
```

Listing 4.2: Use Of The `@Statevar` Annotation

@InterpretationPoint

Methods representing *Default Blocks* in the adaptable software are annotated as `@InterpretationPoint`. Such a method is associated to a part of the behavioural model by a unique identifier. We refer to the beginning of the tagged method itself as *interpretation point*. If the linked part of the behavioural model changes to something else than the default behaviour, the actual body of the annotated method is replaced, and a sequence of different actions is executed, as configured in the behavioural model. Listing 4.3 shows an example, where the registration of new users shall be adaptable at runtime.

```
1 @InterpretationPoint
2 private void registerUserEntry() {
3   // Call the default action.
4   registerConnectingUser();
5 }
```

Listing 4.3: Use Of The `@Interpretationpoint` Annotation

@Action

Methods of the adaptable software that are building blocks of the behavioural model are annotated as `@Action`, which can be composed to model complex behaviour. In other word, these actions are composing elements of the methods (activities) annotated as `@InterpretationPoint`. The actions representing the default behaviour of adaptable software have an identifier that specifies the associated interpretation point, as there can be multiple interpretation points per class. Listing 4.4 shows an example for an action that blocks new users. In a simple scenario where a simple one-to-one replacement of functionality at an interpretation point is performed, an action can be executed, instead of the default behaviour.

```
1 @Action(interpretationPointKey="package.ClassName.registerUserEntry")
2 private void registerConnectingUser() {
3   // Register new users.
4 }
```

Listing 4.4: Use Of The `@Action` Annotation For Default Behaviour

4.5.4 Connecting to the Framework using AOP

One of our main goals is to assure *loose coupling* between the adaptable software and the adaptation framework. To achieve this, the adaptation middleware layer, with its set of state variable adapters, and the model interpreter are glued to the adaptable software using AOP, together with the proposed Java annotations.

GRAF uses JBoss AOP [93], an aspect-oriented framework written in Java. JBoss AOP can be used stand-alone in any Java programming environment, or tightly integrated with the JBoss application server [94]. The described approach can also be implemented

using the static AOP compiler of JBoss AOP, or AspectJ [10], to inject the advice code at compile time. The decision to use JBoss AOP was due to its support for dynamic AOP and its good integration with JBossAS, which makes it feasible to integrate GRAF with enterprise Java applications.

GRAF offers a set of generic *pointcut expressions* that use the GRAF-specific Java annotations. Furthermore, generic advice code is injected into the bytecode of the compiled Java classes that represent or contain adaptable elements. For example, there are pointcut expressions that match all Java fields annotated as `@StateVar`. There is specific advice code for each type of state variable (e.g., `int` or `Integer`). The advice code is part of the *state variable adapters*. Currently, GRAF implements state variable adapters for Java's primitive data types and their corresponding reference types. An example extract of a pointcut expression is shown in Listing 4.5.

```
1 <aspect name="IntegerReificatorAdapter" class="graf.middleware.adapters.
   reification.impl.IntegerStateVariableAdapter" scope="PER_CLASS" />
2
3 <bind pointcut="
4   set(int *->@graf.instrumentation.annotations.StateVar) OR
5   set(int *->@graf.instrumentation.annotations.SyncStateVar) OR
6   set(Integer *->@graf.instrumentation.annotations.StateVar) OR
7   set(Integer *->@graf.instrumentation.annotations.SyncStateVar)">
8   <advice name="toRuntimeModelAdvice" aspect="IntegerReificatorAdapter" />
9 </bind>
```

Listing 4.5: Pointcut Expression For Propagating Integer Values To The Runtime Model On Write Access (Set) Of `@Statevar` And `@Syncstatevar` Fields

Each state variable adapter contains an advice that is injected (i) whenever an annotated Java field is *used* (read-access), or (ii) when it is *defined* (write-access). This enables injecting the necessary code for propagating state variable values from the adaptable software to the runtime model ((`@StateVar`) and `@SyncStateVar`), as well the other way around (`@SyncStateVar`), at class load time. The `@Action` annotation is not used for code injection in the described way. It is mainly used for gathering information about available default and alternative actions when generating a default runtime model, as well as for implementing model interpretation.

4.5.5 Adaptation Rule Engine

GRAF embeds an internal adaptation manager in a form of a rule engine that utilizes first-order logic predicates (adaptation rules) for planning. This rule engine is easy to use and sufficient for many adaption demands and runs concurrently with the rest of the system. It also provides basic mechanisms for adding or removing rules that are represented by the provided interface for adding adaptation rules. The rules can be also viewed and modified at runtime using the provided management extensions.

This event-based adaptation manager implements a provided generic adaptation manager interface and provides a simple way of executing a set of adaptation rules via model manager. The rule engine checks the conditions of an each adaptation rule by querying the runtime model through model manager. If the query evaluates to *true*, the action part of the rule becomes a candidate for execution. The model manager validates the list of candidates and executes the action (transformation), if pre- and post-conditions are met.

4.5.6 External Adaptation Managers

The internal rule engine might be inefficient or incapable of selecting the best possible actions when the application domain is non-deterministic or when software behaviour is highly complex with numerous control variables. In such complex scenarios, the demanded adaptation rule is not trivial, or there might be a conflict between the candidate rules which do not allow adaptation managers to take plausible decision to select the best valid action according to the observed state of the system.

As an alternative solution to the internal rule engine, machine learning techniques can be used as a candidate solution for decision making and planning of SAS. As a proof on concept, a solution for the planning process of adaptation managers based on *reinforcement learning* is developed, and the results are published in [4]. This reinforcement learning based solution explores the action space and learns which actions are more suitable for the system state. Machine learning techniques can be advantages in the scenarios in which there is a pattern (trend) in the changes occurring the domain. In other words, the rule engine is not programmed to deal with each instance of the change to be happened in future, but it can co-relate similar situations and can learn what is a best action to take in each situation.

4.6 Summary

In this chapter, we proposed our model-centric runtime adaptation approach. We presented how, in this approach, runtime models can be considered as connection domains to reduce the complexity of engineering and maintaining an SAS by providing partial and customized representations of the software, tailored to suit the needs of adaptation. We realized this approach by the *Graph-based Runtime Adaptation Framework* (GRAF), which supports model-centric runtime adaptivity based on a combination of querying, transforming, and interpreting behavioral models, as well as state variables, which are causally connected to a software application.

A notable strength of GRAF, which distinguishes it from other adaptation frameworks (e.g., [69, 131, 193, 16]), lies in its ability to support method-level compositional adaptation by interpreting runtime models that are not necessarily at the same abstraction level and expressiveness of the base-layer they are representing. Moreover, GRAF's approach to runtime adaptivity reduces the risk of moving towards self-adaptive software systems.

The next chapter proposes a methodology and a process model to reduce the cost and effort of (re)engineering self-adaptive software systems by minimizing the required changes to make software adaptable without breaking the default functionality.

Chapter 5

Reengineering Towards Model-Centric Self-Adaptive Software

"*A complex system that works is invariably found to have evolved from a simple system that works.*"
–John Gall

It is a well accepted fact that software systems need to constantly adapt in order to be sustainable [112]. The goal of moving towards SAS is to reduce the risk of dealing with future changes. In classic software engineering, adaptive changes are non-corrective changes, which can possibly decrease the cost and effort for subsequent changes. From this perspective, moving towards SAS can be viewed as a preventive software change.

In general, a SAS is a control system in which adaptable software is the *controlled plant* and the adaptation manager is the *controller*. Adaptable software exposes monitoring and sensing data via its sensor interface, and receives control action commands as a set of effecting actions through its effector interface. This synergy is captured by Kephart and Chess as a reference architecture for self-adaptive software systems [98].

In general, engineering controller-based self-adaptive software (SAS) is a three-fold problem of (i) constructing an adaptable software that is observable and controllable, (ii) constructing an adaptation manager for observing and controlling the adaptable software, and (iii) integrating both subsystems into a control system. This is a classic software construction problem, in which we have to develop two sub-systems that interact with each

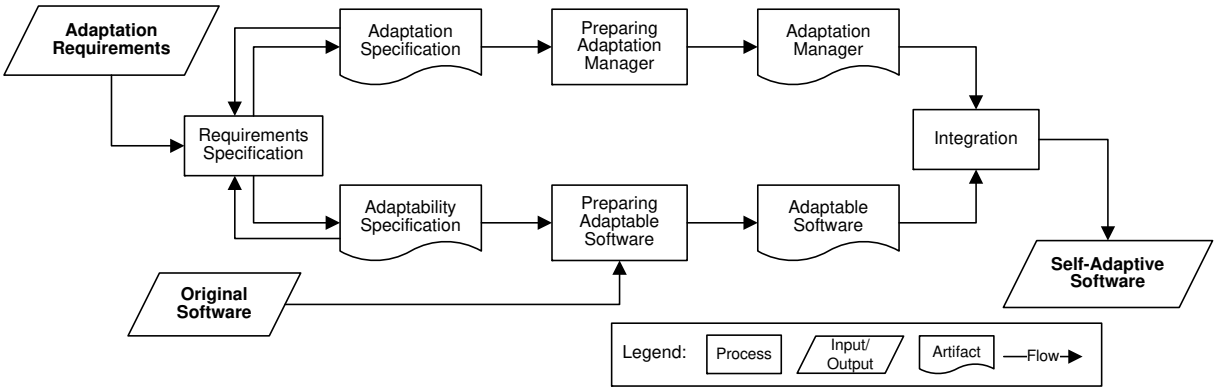

Figure 5.1: A Generic Process Model for Engineering SAS, Conforming to the Conceptual Model Proposed in Chapter 3

other. The process model for building such a system can benefit from a number of successful software methodologies available today. Due to the increasing demand for adaptive systems, and the high costs of developing them from scratch, reengineering current systems into adaptive ones is inevitable.

Considering the conceptual model of SAS problem spaces, discussed in Chapter 3, the process of migrating existing software systems towards SAS can be considered as a process of deriving adaptation and adaptability specifications. Such a process follows two separate paths towards preparing the adaptable software and the adaptation manager, and integrates these components, as illustrated in Figure 5.1. The activities involved in such a process are well known in the field of software evolution and maintenance, and include tasks related to reverse engineering, program comprehension, impact analysis, and change propagation. However, evolution towards SAS also involves other less familiar concerns, such as:

- *Preservation of original behaviour:* The migrated SAS must maintain its original behaviour; that is, if no adaptation is performed at runtime, the SAS must function the same as the software system prior to modernization.
- *Separation of concerns:* The types of changes to be made to the original software (made for the purpose of introducing adaptivity properties) must be kept as separate as possible. Also, such changes should be realized in the form of external components that are loosely coupled with the legacy system.

- *Identification of adaptability factors:* Locating software application fragments that may be involved in runtime adaptation steps requires a good knowledge of the software system and the adaptation requirements.
- *Efficient (cost-effective) evolution process:* The changes to be made to the current software must be kept minimal and cost effective. The types of changes, as well as a systematic procedure for performing each change, must be known in advance.

In order to tackle the above concerns regarding evolution towards SAS, there is a need for a process model that is especially tailored for migrating software systems towards self-adaptivity. In the previous chapter, we introduced GRAF, which provides the required infra-structure and features for supporting such an evolution process. The architecture of GRAF exhibits a loosely-coupled causal connection between the adaptable software and the rest of the framework; this makes GRAF a suitable solution for enabling the efficient incorporation of runtime-adaptivity into existing nonadaptive software systems.

This chapter proposes a reengineering process model for the purpose of preparing and building SAS, for given adaptation requirements, by migrating current (non-adaptive) software. The proposed approach uses GRAF in conjunction with other techniques and tools for achieving this goal. The main objectives of the proposed process are to: (i) reduce the risks and complexities that are normally associated with the modernization of legacy software towards adaptability, and their integration with adaptation managers, (ii) specify the required evolution changes in the form of program transformations, and (iii) preserve the original behaviour of a software system when there is no need for adaptation.

The rest of this chapter describes the process model and provides guidelines on how to assess the associated risks, and to evaluate the effectiveness and fitness of the process model for supporting software changes.

5.1 The Process Model

In Chapter 3, we proposed a conceptual model of the adaptation and adaptability problem spaces to be used for specifying adaptation requirements. We also described how adaptation requirements can be treated as functional system requirements for developing a controller software known as an adaptation manager (AM), and how adaptable software (AS) can be developed based on adaptability specifications, so as to provide the required sensors and effectors.

Recall from Section 3.2 that we can derive adaptation and adaptability specifications from adaptation requirements:

$$(AD \cup Software), S_{Adaptability}, S_{Adaptation} \vdash R_{Adaptation} \tag{5.1}$$

Moreover, in Chapter 4 we introduced a runtime model as a connection domain in a model-centric approach, which represents $AD \cup AS$ and serves as the domain for AM:

$$RM, S_{Adaptation} \vdash R_{Adaptation} \tag{5.2}$$

In GRAF the runtime model is generated, embodied, and maintained by the framework; that is, in the adaptation problem space, the problem of developing the adaptation manager is reduced to the development of appropriate policies (i.e., adaptation rules) for managing the runtime model. Hence, in case of GRAF we have

$$C_{GRAF}, P_{Adaptation} \vdash S_{Adaptation} \tag{5.3}$$

in which $P_{Adaptation}$ is a set of adaptation policies, because the adaptation manager and its rule engine (adaptation mechanism) are parts of the framework and are provided by GRAF. The adaptation policies in the current implementation of GRAF are in the form of ECA adaptation rules running on an embedded rule engine of GRAF, and satisfy $S_{Adaptation}$ by executing queries and transformations on the runtime model. Also, C_{GRAF} is customizable and can be modified to fit specific adaptation needs.

On the other hand, we also need to prepare adaptable software that complies with adaptability specifications. In case of migrating non-adaptive software systems towards runtime adaptability, we start with an operational software that is developed based on a set of specifications (i.e., $S_{Software}$). Any changes to be made on to the software in order to make it adaptable are woven with the current software, and as a result, the original program P will be evolved into a new program P' that satisfies a set of adaptability specifications (i.e., $S_{Adaptability}$). In other words, we will have

$$C, P' \vdash \{S_{Adaptability}, S_{Software}\} \tag{5.4}$$

Here $P' = P + \varepsilon$, where P denotes the properties of the original software and ε denotes the deviation from the original property set. The deviation can be from both functional or non-functional properties of the original software. However, any deviation from the original

functional properties should be prohibited, as SAS must preserve the original behaviour of the non-adaptive software, and the adaptive behaviour should be only triggered by the adaptation manager as required by the adaptation specifications. This means that ideally, ε includes a new set of non-functional properties regarding software observability and controllability. In practice, other non-functional properties can be also affected to some degree; for example, the performance and complexity of the evolved software may be affected as we move towards adaptable software.

Hence, a practical modernization process towards adaptable software has to guarantee that the original behaviour of the software remains intact and deviations from major (and important) non-functional properties do not negatively impact the quality of the new system. One approach for achieving this goal is to allow all types of changes in the software and to perform regression tests to make sure that the original properties are preserved. Although this is a feasible approach, many legacy systems do not have complete (and automated) test cases to be checked as the system evolves, which makes this approach rather costly and impractical.

The second approach is to have an evolution process, in which evolution changes are limited by a set of transformations that maintain the original behaviour of the software. This is the approach followed in our proposed process model, in which the adaptable software is the refactored version of the original software. Having transformations in the form of refactorings will guarantee that the functional properties are maintained and only non-functional properties will be affected; hence, for the functional properties we can have $P' \simeq P$. Therefore, using GRAF and its infrastructure for adding sensors and effectors dynamically shifts deviations in the satisfactions of non-functional properties from the program to the computer. Equation 5.4 can be reformulated as

$$C_{GRAF}, P' \vdash \{S_{Adaptability}, S_{Software}\} \tag{5.5}$$

in which $P' \simeq P$ and C_{GRAF} ensures the integration and operation of adaptable software with GRAF. In a nutshell, in the GRAF approach, software adaptability is achieved by transforming the original software elements into adaptable elements that are exposed via sensor and effector interfaces, and are constituent elements of the runtime model. By using this approach and defining the required evolution changes towards adaptability as a set of transformations, we argue that *software is evolvable towards adaptability, if there exists a sequence of refactorings from the original software to the required adaptable software.*

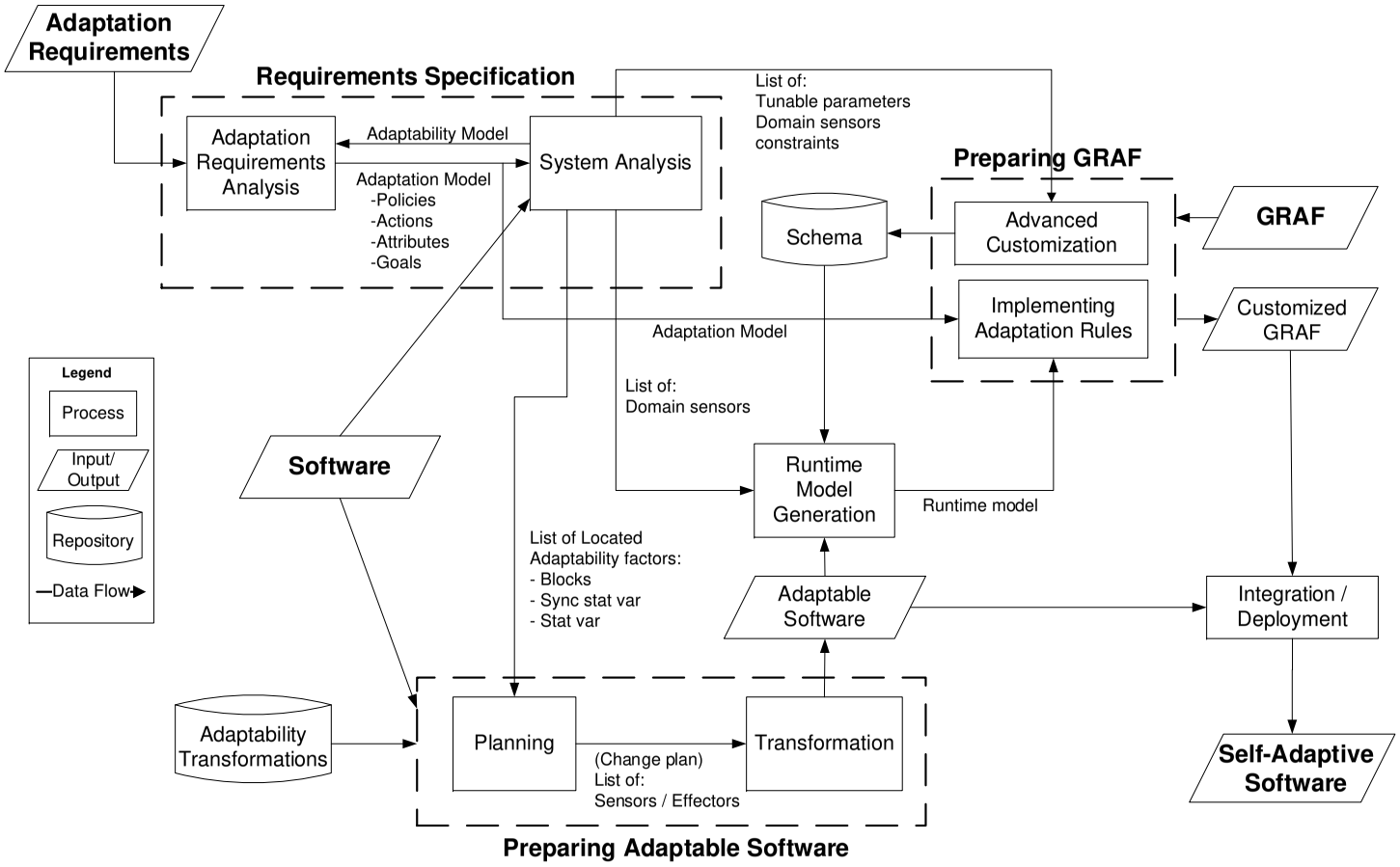

Figure 5.2: The Process Model

Based on the above definitions and design considerations, our process model for building GRAF-based model-centric SAS is illustrated in Figure 5.2. The proposed process model uses the domain-driven design approach [61], which advocates a deep connection between the software implementation and an evolving model of the core business concepts. The inputs of the proposed process model are original software, the set of adaptation requirements, and GRAF; and its outputs are the adaptable software, and a customized GRAF that is able to control the adaptable software.

The modernization of current systems towards adaptivity requires additional steps in terms of program comprehension and necessary transformations of the original application. Hence, the process incorporates an additional analysis step for investigating and extracting adaptability artifacts in the original software. In addition, the preparation of the adaptable software is based on a set of transformations for adding the required adaptability. The upcoming sections will elaborate this process model in detail and highlight the challenges associated with each step.

5.2 Adaptation Requirements Analysis

The order of capturing and modeling entities in the adaptation and adaptability problem spaces is debatable. This is mainly because of the dependencies among these spaces. In the proposed process model, given a set of adaptation requirements, the initial step is to derive the adaptation specification model. This model is used as input for system analysis, and is later on used to construct the adaptation manager. However, the overlap between adaptation requirements analysis and system analysis varies as the process progresses: initially, there is very little analysis done on the application domain and software, and the majority of the effort goes toward gathering[1] and modeling requirements.

The adaptation manager in GRAF inherits its behavioural model from the framework. Therefore, the behaviour specifications of the adaptation model of GRAF-based SAS are adaptation policies; these policies are to be implemented as event-condition-action (ECA) rules (adaptation rules) that specify exactly what to do in a given state. In this case, the adaptation behaviour is essentially compiled as a set of adaptation rules in the form of logical predicates. Adaptation rules can be used to check deviations from adaptation goals, or to trigger appropriate actions.

[1]In this thesis, we assume that the adaptation requirements are provided as the starting point for engineering SAS, and the process of elicitation or derivation of adaptation requirements from stakeholders or system artifacts is beyond the scope of this thesis.

The event and condition parts of each adaptation rule require a set of domain attributes. These domain attributes are used to determine the state of the application or that of its operating environment. Adaptation rules also have an action part, which are to be executed by effectors on the adaptable software. At this stage, the effectors are not yet explored; however, we are concerned with the expected result of executing actions via appropriate effectors. This means that after executing the selected action, the software should exhibit the adaptive behaviour required. However, the means by which the requested behaviour is achieved is not defined at this stage, before analyzing the software's internals. The pre and post conditions of each action are also represented in terms of domain attributes, the value of which must be also available to the adaptation manager.

The details involved in engineering adaptation requirements is beyond the scope of this thesis. However, there are a number of well-established requirements engineering approaches (e.g., KAOS by van Lamsweerde [189]) that are helpful in eliciting and specifying adaptation policies. Among them, the *quality-driven framework for engineering an adaptation manger (QFeam)* by Salehei [160] best fits our goals for elicitating and analyzing adaptation requirements. QFeam is mainly designed for engineering the planning process (i.e., detecting and decision making) of SAS and includes an *adaptation requirements analysis* sub-process for modeling the entities of the adaptation problem space; specifically, the process supports modeling, the adaptation goals, domain attributes and adaptation actions, all of which can be used for specifying the initial adaptation model of a given set of adaptation requirements in our process model.

Adaptation modeling results in an initial model of the required sensors and effectors to be provided by the adaptable software and the application domain. This model is subject to change before a finalized runtime model is developed.

5.3 System Analysis

When migrating software towards runtime adaptivity, analyzing the system to be changed is a mandatory step towards gaining a clear understanding of the software and its application domain. Moreover, in the adaptation problem space, this process can be also viewed as a part of the domain-analysis procedure for providing domain knowledge to support *adaptation requirements analysis* process.

During this process, the developer analyzes the software with regards to the developed adaptation model, and identifies the concepts that are required to be monitored by sensors and controlled by effectors. These concepts are then match to relevant software elements as

the required adaptability factors. If the required concepts cannot be identified or matched to software elements, they should either be shared by the application domain, or the adaptation model needs to be adjusted to meet the new domain constraints. For the identified factors, an additional analysis step is required for investigating the feasibility of runtime adaptation with respect to their exposure and change. Finally, we match the concepts to adaptability factors of the adaptability model by creating traceability links between them.

The result of the software analysis step is a preliminary software-adaptability model in the form of a list of located *default block*, *state variable*, and *Sync State Variable* adaptability factors. In the case that sensors (and possibly effectors) over an application domain are needed, an additional list is allocated to these sensors and effectors. Implementing external sensors and effectors that can directly monitor and effect the application domain is beyond the scope of the modernization process; however, during system analysis, we develop and maintain a list of these sensors and effectors, because the implementation of the adaptation manager will depend on them, and external management interfaces must be provided by GRAF to support them.

5.3.1 Locating Concepts

A software system operates in the application domain. The behaviour and interaction of this system with its domain can be observed and controlled in different ways. This is why the placement of sensors, effectors, and phenomena are critically important in any control system, including control-based SAS. The location of these items determines whether an open-loop or closed-loop control system is used, and whether a domain model is needed. For example, if we can only observe the application's input, we need a model of the software system to determine the system's output givem the observed input. In control systems, the controller only has access to the system's output, hence, the plant is observable only if we can determine its state from the system's output [141]. Therefore, in cases where the software is not naturally observable, a practical solution is to use sensor data as additional output data in determining the system's state.

In order to observe a change in software's operating environment, by monitoring the software, we have to make sure that the software can observe the change happening in its application domain. If the change is not propagated to the software, there will be no way to observe that change from within software, unless phenomena that store change information are shared with the software. In other words, a change that does not alter the application's state cannot be observed by the adaptation manager. A similar argument can

be made for controllability: a change should alter the output (i.e., it should be observed from the world, for both functional and non-functional properties); otherwise, the change is useless. Therefore, it is essential to create a mapping between the phenomena in the AD, which are described by the AR, and the corresponding phenomena in the AS, as abstractly illustrated in Figure 5.3. A possible solution is to explore this unknown space (from a user's perspective), and to identify shared phenomena between the software and the AM by creating a mapping from shared phenomena in the application domain to the ones in the original software.

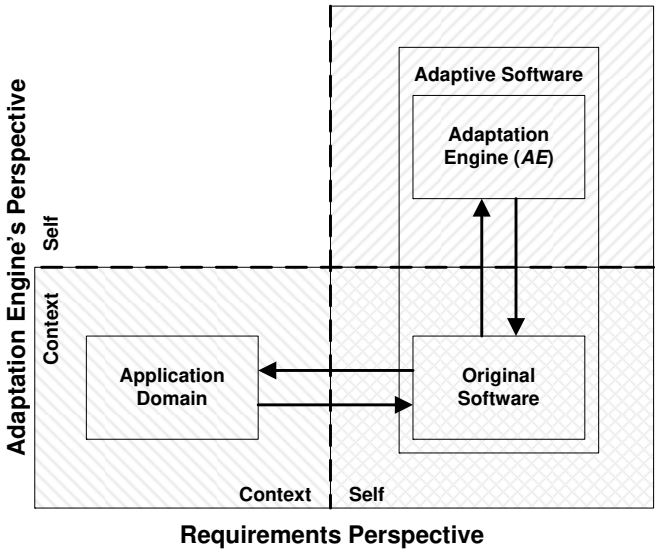

Figure 5.3: Problem of Finding a Mapping from Phenomena in the AD, Which are Described by the AR's, to Their Corresponding Phenomena in the AS

The following example demonstrates the idea of mapping phenomena from the AD to those in the AS. An AR of a sample Internet Protocol Television (IPTV) application might be to "alter service quality in the case of high system load". In this situation, the AD's phenomena to be observed by the AM is "system load", and the AD phenomena to be effected is "service quality". Moreover, in the AS, the two AD's phenomena might be represented by (i.e., mapped to) the "Number Of Active Sessions" and "Service Type" state variables. These two variables are phenomena in the AS, and can be shared with the AM through sensor and effector interfaces.

As in GRAF, state variables are reified to and reflected from the runtime model to provide parameter adaptation. Compositional adaptation is achieved by composing and

interpreting activities at runtime. The concepts to be identifies at this stage are the *default block* and *state variable* adaptability factors for the required sensors and effectors. Therefore, further analysis is required to comprehend the feasibility of sharing each concept, and the impact of changing the identified adaptability factors.

5.3.2 Comprehending Software Adaptability

A software system possibly includes concepts that are required by the adaptation manager but cannot be provided by the software, for example, due to the high costs of the changes required for including them. The whole change cycle can be viewed as a co-evolution change problem between the software and its adaptation manager, where the goal is to come up with the specification that best satisfies the adaptation requirements. As described in Subsection 3.4.2, application-level adaptation is either achieved by changing the application's control flow or its state. Moreover, to observe the changes in the operating environment and software, we have to monitor the changes in the application state. Therefore, this step can be divided into two parts, one for comprehending each of the application's state variables and control flow.

State Variable Analysis

We have to analyze the related *state variables* of a software system in order to comprehend:

- Which elements of the software are affected by changes in the environment, including the changes in control flow? These elements reflect the changes in the environment, but they are not necessarily part of the software's input/output phenomena.
- Among these elements, which variables (or sets of variables) represent state transition as the change takes place?
- What are the initial states of adaptation scenarios?
- What are the type, range (coverage), change frequency, aliases (i.e., other variables that hold the same state information), and accessor methods?
- When does the change become observable? This is highly dependant on the nature of the change to be observed; in most cases, the change is observable as it happens in the environment. However, there are other scenarios in which the occurrence of a change can be predicted based on the trend of other changes. In some other cases, it takes time to for a change to the affect system state.

Control Flow Analysis

For analyzing the impact of changes and identifying code blocks (*default blocks*) that are responsible for providing the desired behaviour, we have to determine:

- The effect of the change on the system's behaviour.
- Whether the change in behaviour is stateful. That is, in the case that the change subsides, will the system regain its expected behaviour.
- Whether the change alters the application's control flow
- The impact of the change on response time at different levels; e.g., end-to-end method and data access.
- The impact of the change on system resources.
- The relationship between non-adaptive and adaptive behaviour.
- Which elements are involved in providing the required adaptive behaviours.
- Whether it is possible to reuse the constituent elements for the purpose of switch behaviours.
- In case of reuse, whether we need to re-orchestrate the constituent elements, or just adjust them properly.

These analysis steps will also reveal information regarding the variation degree, pre and post requirements for the activation and operation of each sensor and effector, and other constraints and invariants associated with each software element related to adaptation. Employing this domain knowledge into the development process of the adaptation manager is crucial for the performance and stability of the final system.

However, for some missing adaptability factors, it might be impossible (e.g., due to the high costs of change) to create a mapping. In these cases, we have to modify the list of adaptability factors. Changing the list of adaptability factors results in a change in the adaptation specifications, which are the interface specifications between the adaptable software and the adaptation manager. Finally, the change will be propagated to the adaptation model and the adaptation manager. This basically means that adaptation manager must evolve in order to fit an adaptable software that can be feasibly prepared.

For example, assume the adaptation requirement of reducing the response time of a feature implemented by a set of methods. The problem is that no software element holds data regarding response times. Here, a possible solution is to add a new member-variable `reponseTimeX` and set its value by measuring the difference between the method's start-time and end-time as shown in Listing 5.1.

```
1 public class A {
2   @StateVar
3     long reponseTimeX = 0;
4   private void methodX(){ // ...
5     long currentTime = System.currentTimeMillis();
6     // ... method body
7     reponseTimeX  =  System.currentTimeMillis() - currentTime;
8   }
9 }
```

Listing 5.1: Making Data Explicit to be Used as State Variable

By aggregating the response time for all methods that implement the feature, plus other delays to be estimated by the software and domain model, we can estimate the required end-to-end response time. Although this solution does not provide the response time from a user's point of view (i.e., end-to-end response time), it provides a proxy for this data by locating, mapping, and sharing a set of phenomena in the software. If we can adjust adaptation policies to accept and utilize this data instead of the specified one, we can have successfully map from a application domain phenomena to software elements.

5.3.3 Comparing Adaptability Models

In the GRAF approach, there is no need to match the complete list of adaptability factors (see Table 3.3). Some of the factors will be automatically matched according to their dependencies with other factors. The required adaptability factors are listed in Table 5.1. Based on how an adaptability factor is supported by the current software elements, it is marked as being either *missing*, *available*, *located*, or *tagged*:

- *Missing* factors require the introduction of additional elements.
- *Located* factors are software elements that can serve as an adaptability factor, but must be modified to do so.

- *Available* factors are those that are assigned to existing program elements.
- *Tagged* factors are available factors that are annotated, so to be exposed and shared by GRAF.

For missing adaptability factors, new software elements must be introduced at several levels of granularity. Since GRAF supports adaptation on the basis of member variables and methods, new software elements have to be introduced at this level. The *system analysis* sub-process answers the questions of *"what we currently have"* in terms of the required adaptability in current software, and *"what we can have"*. This sub-process also gives a plan for preparing the target system for the next step, which answer the questions of *"how we can have the required adaptability factors"*. The next section describes how to plan for changing a software system to provide *located* and *missing* adaptability factors.

5.4 Planning

The outcome of *system analysis* is a list of located *default block*, *state variable*, and *sync state variable* adaptability factors. Each of these located factors can be realized by different sensing and effecting styles. As discussed in Subsection 3.4.5, each style has advantages and disadvantages, and the choice of style has to be made on a case by case basis.

Considering adaptation requirements as a set of change requests, in all scenarios, there is more than one way to satisfy the required adaptive behaviour by changing the software. Different combinations of sensors and effectors can be used to fulfil the same change request, and in some cases, the only feasible option is to change the software statically as a classic adaptive-maintenance change. The system designer must investigate alternative solutions and their pros and cons, and select the most appropriate change plan.

In general, three important factors play key roles in the quality of software change: precision, performance and flexibility. Moreover, another important factor that affects software change is the cost and effort required for performing the change. These properties form a multi-criteria optimization problem, as there is always a trade-off among these properties. Each valid solution has a different impact on these non-functional properties, each with a unique set of advantages, disadvantages, and risks associated with different stages of the software life-cycle. By evaluating the tradeoffs, and by checking application's mission and its non-functional requirements, a developer must select the best solution. For example, a design decision has to be made on the binding time for performing software

Table 5.1: List of Minimal Required Adaptability Factors in GRAF Approach

Change	**Adaptability Factor**	**Matched Element**	**Req.**	**Reason**
Flow	Replace Block::Default Block	Method	Yes	
	Replace Block::Alternative Block	Method	No	Composed and interpreted by GRAF
	Replace Block::Default Action	Method	Yes	
	Replace Block::Alternative Action	Method	Yes	
	Replace Block::Turning Point	Conditional Exp.	No	Part of Default Block, extracted or created automatically by GRAF
	Replace Block::Condition	Variable	No	Part of Default Block, extracted or created automatically by GRAF
	Select Block::Default Block	Code Block	Yes	
	Select Block::Alternative Block	Code Block	Yes	
	Select Block::Default Action	Method	No	Part of Default Block
	Select Block::Alternative Action	Method	No	Part of Alternative Block
	Select Block::Turning Point	Conditional Exp.	Yes	
	Select Block::Condition	Variable	Yes	
State	Direct Change::State Variable	Variable	Yes	
	Indirect Change::State Variable	Variable	No	Use Direct Change instead
	Indirect Change::State Definer	Method	No	Use Direct Change instead
	Direct Monitor::State Variable	Variable	Yes	
	Indirect Monitor::State Variable	Variable	No	Use Direct Monitor instead
	Indirect Monitor::State Definer	Method	No	Use Direct Monitor instead
	Indirect Monitor::State User	Method	No	Use Direct Monitor instead

change; whether, the instrumentation and activation of sensors is performed statically or dynamically.

The variety of design options for each sensor and effector calls for a planning process to (i) investigate alternative solutions, (ii) select the optimum solution, (iii) explore the possible design patterns than can be used to improve software quality and to facilitate the transformations, and (iv) prepare a minimal set of changes that provide all of the required sensors and effectors. The rest of this section gives an overview of the main design decision, that need to be made at this stage of the modernization process. The result of this planning process is a list of all software elements that have to be changed to meet the specifications of the desired adaptability model.

5.4.1 Addressing Variability for Adaptation

Variability in runtime behaviour is generally achieved by either *parameter adaptation* or *compositional adaptation* [125]. In parameter adaptation, all of the code segments required for providing the required adaptive behaviours are available and already coded in the software. In this approach, variation is achieved by setting a control parameter to select the desired behaviour (or setting) among the available choices. This approach requires the interpretation of the behaviour, because of all the required semantics are compiled into the program. On the other hand, compositional adaptation allows adaptive behaviour to be generated and composed dynamically at runtime. This gives more flexibility as the composing elements can be put together in different valid compositions. However, this approach requires code generation or interpretation at runtime, because each valid composition negatively impacts adaptation performance. In our adaptability model, the *replace block* effector style supports compositional adaptation and *select block* style applies to parameter adaptation. However, *select block* has to be paired with a state-variable control style, so as to model and expose the *decision criteria* as a *sync state variable*.

Figure 5.4(a) represents the flow graph of a non-adaptable software application, with a missing alternative action for replacing the default behaviour provided, with the available default action at node 2. After adding the missing alternative action, the actions can be swapped following one of the *parameter-adaptation* or *compositional adaptation* approaches.

In *parameter adaptation*, the application's control flow can be redirected to an alternative action. This is done at a *turning point*, which is a branch in the control flow with a well-defined decision criterion. This condition is realized using a synchronized state variable that can be tuned by GRAF. In Figure 5.4(b), the original graph is augmented by an

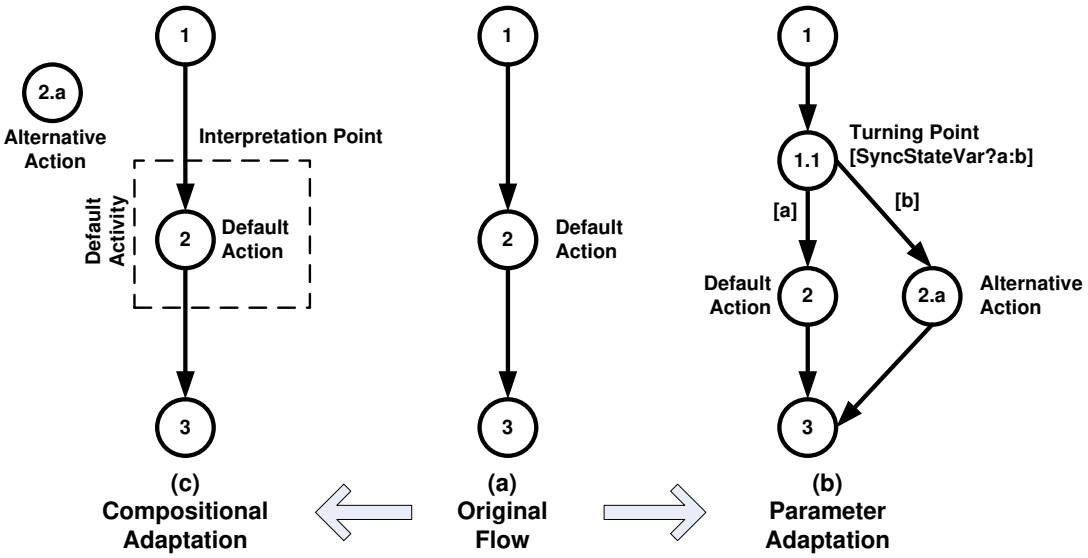

Figure 5.4: Modifications Towards Runtime Adaptation

alternative action at the turning point (node 1.1). The value of the state variable at node 1.1 can be adjusted by the adaptation framework to select either the default action (node 2) or the new alternative action (node 2.a).

For *compositional adaptation*, the adaptation framework replaces a block of the adaptable software's control flow with the execution of the alternative behaviour at runtime. Figure 5.4(c) illustrates how this change takes place. The rule engine is able to replace node 2 with node 2.a in a behaviour description that is part of the runtime model. Node 2.a has to satisfy the pre-conditions and post-conditions of node 2. Here, the located *default block* adaptability factor has to be extracted as a new method. This factor is bound to an activity in the behavioural model that describes the actions to be executed using the available actions. At runtime, model interpretation takes place when the control flow reaches the *interpretation point*.

5.4.2 Deciding on State Variables Access Mechanisms

GRAF provides direct access to *SyncStateVars* variables. Using this approach, any field variable can be denoted as a state variable, which can be set at runtime. However, in some situations, direct access is not the best solution and can even be considered as a bad code

smell. For example, (i) changes to the exposed state variable may be only desirable during a limited time period (e.g., after class construction, for the purpose of self-configuration), (ii) the state variable can be a one-time mutable attribute that cannot be declared final, or (iii) the tuning of the state variable is part of the compositional adaptation. In these scenarios, or other similar cases, access to state variables should be limited. One solution is to have additional policies to check for these constraints as part of the pre-conditions for each action; another solution is to implement the constraints as model-level constraints in a runtime model schema. There is also a third approach, which is to not expose the attribute directly as a *SyncStateVar*, but to provide indirect access to it by encapsulating it with accessor methods. The advantage of this solution is that when the data and behaviour that uses the attribute are clustered together, it is easier to change the code, because the changed code is in one place rather than scattered throughout the program (within different methods that also *define* the state).

The same argument and solution also applies to direct and indirect state monitoring. There are several reasons for limiting the state monitoring process. For example, (i) the application may be in a transient state and we may need to temporarily stop the sensing process, (ii) we may need to sense the state during behaviour interpretation, or (iii) we may need to enable/disable sensors selectively based on their location in the code.

5.4.3 Anticipating Future Changes

Planning for evolution towards SAS should not be done only to address current adaptation needs, but should be also extended to support changes that might happen in the future. Hence, a clear vision is needed for changes that will happen in the future. The risk of moving towards SAS will not pay off if there is no visibility about potential upcoming changes. For example, consider a scenario in which the rate of environmental changes that demand an adaptive behaviour is either too low or too high: if the probability of the change is extremely low, the development and maintenance costs for having SAS will not pay off in the long term; and if the change is too frequent, it might be more appropriate to statically evolve the software to support the change, and forbid adaptation costs.

It may be impossible to predict upcoming change requests for a new software. However, in the case of legacy software and software systems that have lived for years, this prediction is possible with an acceptable accuracy by considering historical data associated with software change. A possible way to make such a prediction is to analyze the software repository and to identify change patterns and models that support change prediction. We realize this solution as the Neural Network-based Temporal Change Prediction (NNTCP)

framework [5]. The framework indicates "where" the changes are likely to happen (i.e., hot spots) in the software, and then adds the time dimension to predict "when" the changes may occur. This information can be used to estimate the rate of future required changes in software, so as to support managers and developers in planning evolution and adaptation activities more efficiently.

The approach uses the history log of software changes from a software repository to measure applicable development metrics. The metrics are fed into an artificial neural network that estimates the future change date of a given entity. More details on the framework, including the conducted case studies for the proof of concept, are given in [5].

5.5 Preparing Adaptable Software

At this stage, we will evolve source code such that its elements can be clearly mapped to their relevant adaptability factors. As the change plan suggests, missing factors shall be included in the software, and to be usable, the located factors will require changes in the software. These evolutionary changes are transformations of the original software that enable establishing a complete mapping between the correlated elements, where the transformations preserve the default behaviour of the software.

The transformations are defined in a hierarchial model, allowing us to implement complex transformations as a composition of basic refactorings. The *refactorings* are transformations of the program, which are a manipulation of source code that preserves the program's behaviour [142]. The refactorings (and the analysis preceding them) require a programmatically manipulable representation of the developed system, and since they are behaviour preserving, their composition as complex transformations is necessarily behaviour preserving. Adaptability transformations also use *creational transformations* to add missing software elements (i.e., adaptability factors), that are required by sensors and effectors.

Moreover, the quality and performance of adaptable software can be further enhanced by using several design patterns. The application of these design patterns can be supported by transformations too, which is extensively addressed in the literature [100, 127, 186]. Thus, in this research, we only discuss adaptability transformations for incorporating sensor and effector styles in software. For other transformations, we provide references to sources that describe them, while remarking on their optional application in adaptability transformations.

5.5.1 Refactorings

The refactorings used to support our transformations are proper functions without side effects on the program to include *missing* and *located* adaptability factors. These basic refactorings are classic and part of available refactoring suites [65, 127], which are also automated and supported by transformation tools and languages [127, 134]. Table 5.2 shows a list of the 25 refactorings, used to support *Adaptability Transformations*. The list contains a selection of refactorings from *Refactoring: Improving the Design of Existing Code* book by M. Fowler [65], and one refactoring, *Convert Local Variable to Field*, which is a refactoring introduced since Eclipse 3.4 to support extracting field variables [60]. Appendix A gives a brief description to each refactoring. The refactorings are labeled with an ID, and are grouped into three categories:

- *Primitive*: refactorings that shape the main transformation process of adaptability transformations.
- *Supporting*: refactorings that provide infrastructure for other refactorings and transformations.
- *Improving*: refactorings whose usage is optional, but which can be used to improve the quality of sensors and effectors.

As with the listed refactorings, the transformation process can also benefit from *refactorings to patterns* [100] to impose design patterns which directly or indirectly contribute to adaptability by addressing the problems and difficulties associated with including sensors and effectors. However, some of these design patterns (e.g., *Micro Kernel*) are high-level, making them only applicable to creating software from scratch. Some other patterns are technology and platform dependent (*e.g. Declarative Component Configuration*, *Component Configurator*, and *Activator*). Although applying these patterns is not impossible on legacy software, the changes to prepare the required infrastructures are extensive and in most cases not cost effective. Thus, in our modernization process, we only consider the refactorings listed in Table 5.2.

Meanwhile, other applicable design patterns and their supporting transformations can assist developers to: (i) increase software maintainability to facilitate current and future evolution changes, and (ii) contribute to performance or flexibility of adaptation changes, can be used to extend the provided list of refactorings. For example the *Visitor* pattern can be used to create a concrete instance of a visitable object for each data type in *conditional* by creating a concrete instance of a *Visitor* class that encapsulates logic of each *conditional*.

Table 5.2: List of Selected *Refactorings (RF)*

Category	**ID**	**Name**
Primitive	RF_3	Encapsulate Field
	RF_5	Extract Method
	RF_{20}	Replace Temp with Query
	RF_{21}	Separate Query from Modifier
	RF_{23}	Substitute Algorithm
Supporting	RF_7	Inline Temp
	RF_8	Introduce Assertion
	RF_9	Introduce Explaining Variable (Extract Local Variable)
	RF_{13}	Reduce Scope of Variable
	RF_{16}	Replace Exception with Test
	RF_{17}	Replace Method with Method Object
	RF_{22}	Split Temporary Variable
	RF_{24}	Convert Local Variable to Field
Improving	RF_1	Consolidate Conditional Expression
	RF_2	Consolidate Duplicate Conditional Fragments
	RF_4	Decompose Conditional
	RF_6	Form Template Method
	RF_{10}	Introduce Parameter Object
	RF_{11}	Parameterize Method
	RF_{12}	Preserve Whole Object
	RF_{14}	Replace Array with Object
	RF_{15}	Replace Conditional with Polymorphism
	RF_{18}	Replace Nested Conditional with Guard Clauses
	RF_{19}	Replace Parameter with Explicit Methods

Alternatively, we can use the *Strategy* pattern to define a family of algorithms, encapsulate each one, and make them interchangeable at runtime via *Select Block* and *Change State* effectors. Strategy lets the algorithm vary independently from the clients using it, by capturing the abstraction in an interface and moving implementation details to derived classes. Another option is to combine the *State* pattern with *Change State* and *Monitor State* by allowing an object to alter its behaviour when its internal state changes.

5.5.2 Creational Transformations

Along with basic refactorings, we also need a new set of transformations for adding missing adaptability factors. These transformations are *additive* and have a creational nature, which means that they all add a specific code block to the application. The location of *missing* factors depend on the location of the associated sensors and effectors. Each helper function is specified by its name, return type, argument types, and provides a brief textual description of its purpose. Table 5.3 shows the complete list of creational transformations used to prepare adaptable software.

Table 5.3: List of *Creational Transformations (CT)*

ID	**Name**	**Adaptability Factor**
CT_1	addBranch(block)	SelectBlock::TurningPoint
CT_2	addBlock(entrypoint, exitpoint)	SelectBlock::AlternativeBlock
CT_3	addField(class, identifier, type, modifiers, initial Value)	SelectBlock::Condition DirectMonitor::StateVariable
CT_4	addMethod(class, identifier, type, modifiers, parameters)	ReplaceBlock::AlternativeAction

- *addField* and *addMethod* creational transformations basically add new member variables and member functions to a specified class. Their implementation is programming language dependant, but for all object-oriented languages the process is straightforward.
- *addBlock* surrounds a set of statements with appropriate keywords or literals to form a block. In Java the block is defined by enclosing the code in {}. Local variables defined inside {} are not known outside the block, though they are allocated when the method starts, not when the block is entered. This means during addBlock we may need a refactoring to move variable definitions out of the block.

- addBranch() adds a branch at the starting point of a block, clones the block in all branch legs, and sets the condition expression. An alternate implementation is to just add the block to one of the branch legs and set the condition accordingly to always navigate the flow to it. The implementation of this transformation is based on conditional constructs to selectively alter the control flow based on some condition. Most structural programming languages support different types of `IF` and `SWITCH` conditional statements (a.k.a. Selection statements) for this purpose. Here, we use the If-Else Construct, but the transformation can be customized to use other conditional statements as well. The if-else statement provides a selection control structure to execute a section of code if and only if an explicit run-time condition is met. The condition is an expression which evaluates to a boolean value, that is, either true or false.

5.5.3 Tagging Transformations

Tagging elements with annotations allow us to create traceability links between adaptability factors (composing elements of the adaptability model) and software elements. GRAF also needs a mapping between runtime model elements and their respective source code elements to initialize its runtime model. This initial runtime model is also generated from annotated software elements. The connection between adaptability factors and the predefined sensor and effector interfaces of GRAF (see Section 4.2) is established by instrumenting the program elements by Java annotations. The annotated elements of the adaptable software will be automatically wrapped by appropriate aspect code to expose the interfaces.

We annotate matched elements for later instrumentation using custom Java annotations. The actual instrumentation is performed at load-time of classes, using dynamic AOP. Each source code annotation is used for load-time instrumentation according to an AOP pointcut, which will be used to inject either state variable adapters or the model interpreter hooks into Java classes. The implementation details are discussed in Subsection 4.5.4.

5.5.4 Adaptability Transformations

GRAF approach uses a set of high-level transformations to enable each type of sensing and effecting styles. The goal of using adaptability transformations is to remove the burden of tedious and error prone code reorganization from the developer by providing a systematic

Table 5.4: List of *Tagging Transformations (TT)*

ID	Annotation	Adaptability Factor
TT_1	@InterpretationPoint	ReplaceBlock::DefaultBlock
TT_2	@action(default)	ReplaceBlock::DefaultAction
TT_3	@action	ReplaceBlock::AlternativeAction IndirectAccess::StateDefiner IndirectMonitor::StateUser
TT_4	@SyncStateVar	SelectBlock::Condition DirectAccess::StateVariable
TT_5	@StateVar	DirectMonitor::StateVariable

guideline on how to use *Refactorings*, *Creational Transformations*, and *Tagging Transformations* to prepare adaptable software. The list of *Adaptability Transformations*, and their composing transformations is shown in Table 5.5.

Table 5.5: List of *Adaptability Transformations (AT)*

ID	Name	Composing Transformations
AT_1	Add Replace Block Effector	$\{CT_4\}$ $\{TT_1, TT_2, TT_3\}$ $\{RF_3, RF_5, RF_6, RF_7, RF_{10}, RF_{12}, RF_{13}, RF_{16}, RF_{17}, RF_{20}, RF_{23}\}$
AT_2	Add Select Block Effector	$\{CT_1, CT_2, CT_3\}$ $\{TT_4\}$ $\{RF_1, RF_2, RF_4, RF_{13}, RF_{15}, RF_{18}, RF_{19}, RF_{23}\}$
AT_3	Add Change State Effector	$\{TT_4\}$ $\{RF_9, RF_{14}, RF_{22}, RF_{24}\}$
AT_4	Add Monitor State Sensor	$\{CT_3\}$ $\{TT_5\}$ $\{RF_5, RF_8, RF_9, RF_{14}, RF_{21}, RF_{22}\}$

The presented adaptability transformations are able to add all sensors and effectors required by GRAF based on the sensing and effecting styles. Each of the adaptability transformations has the following structure:

- **Requires:** They must hold in order to be able to apply the transformation.
- **Transformation Composition:** Formalizes adaptability transformations as regular expressions. Each transformation has three sub-expression corresponding to preparation, main, and refinement processes. The complete transformation is a conjunction of all three sub-expressions. The following standard operations are used to construct the regular expressions:

  - '|' for boolean "or" that separates alternatives (e.g., gray|grey can match "gray" or "grey").
  - '(' ')' for grouping and to define the scope and precedence of the operators (e.g., gray—grey and gr(a|e)y are equivalent patterns, which both describe the set of "gray" and "grey").
  - Quantifiers after a token or group to specify how often such preceding element is allowed to occur:
    - '?' indicates there is zero or one of the preceding element (e.g., colou?r matches both "color" and "colour").
    - '$*$' indicates there is zero or more of the preceding element (e.g., ab$*$c matches "ac", "abc", "abbc", "abbbc", and so on).
    - '+' indicates there is one or more of the preceding element (e.g., ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac").

- **Preparation Process:** list of supporter transformations to be done if applicable to prepare program elements for the main transformation process.
- **Main Process:** This is a concise, step-by-step description on how to carry out and implement transformations.
- **Refinement Process:** Set of optional refactorings to improve the quality of the transformed elements.
- **Considerations:** There are several consideration that has to be taken into account before, during, and after the transformation process that are listed in this section.

AT_1: Add Replace Block Effector

Requires A located block (i.e., *default block*) that implements the default behaviour that has to be changed at runtime.

Transformation Composition

- **Preparation:** $((RF_7)|(RF_{13})|(RF_{16})|(RF_{17}))^*$
- **Main:** $(RF_3)(TT_1)((RF_3)|(RF_5))^*(RF_{20})^*((RF_3)(TT_2))^+((CT_4)(RF_{23})(TT_3))^*$
- **Refinement:** $((RF_6)|(RF_{10})|(RF_{12}))^*$

Preparation Process

- RF_7: Remove temp variables that are assigned to once with a simple expression by replacing all references with their defining expression.
- RF_{13}: Reduce the scope of the variables that are defined and used inside the block, which have a scope that is larger than where it is used. This refactoring can possibly reduce the number of the parameters required for default actions.
- RF_{16}: Remove the exceptions that are thrown from within the block by substituting them with test expressions.
- RF_{17}: If the block is long and uses local variables in such a way that you cannot apply Extract Method, then turn the method into its own object so that all the local variables would become fields on that object.

Main Process

1. RF_3: Extract block as a new method.
2. TT_1: Tag the new method with @InterpretationPoint.
3. RF_3 or RF_5: Substitute assignments with setter() method calls.
4. RF_{20}: Replace temp variables with query method calls.
5. RF_3: Extract composing actions of the block as new methods, if there is a set of consequent actions that resemble a meaningful behaviour (i.e., actions).
6. TT_2: Tag all extracted actions with @action(default).
7. CT_4: Add missing alternative actions for each default action.
8. RF_{23}: Change the body of the alternative actions according to their corresponding default action.
9. TT_3: Tag all alternative actions with @action.

Refinement Process

- RF_6: Break down the method body as a set of composing action blocks and decision nodes.
- RF_{10} or RF_{12}: Reduce number of parameters passed to a default action and all its related alternative actions.

Considerations

- Each alternative action's contract must conform to its corresponding default action's contract.
- RF_{12} (refinement process) requires that alternative actions to be refactored are not in use (they are not borrowed from other parts of program).

AT_2: Add Select Block Effector

Requires A located block (i.e., *default block*) that implements the default behaviour that has to be changed at runtime.

Transformation Composition

- **Preparation:** $(RF_{13})^*$
- **Main:** $(CT_2)((CT_3)(TT_4)(CT_1)(RF_{23}))^+$
- **Refinement:** $((RF_1)|(RF_2)|(RF_4)|(RF_{15})|(RF_{18}))^*|(RF_{19})$

Preparation Process

- RF_{13}: Reduce the scope of the variables that are defined and used inside the block, which have a scope that is larger than where they are used. This refactoring may reduce the number of the parameters required for default actions.

Main Process

1. CT_2: Add the located block.
2. For all required alternative behaviours of the located block:
   (a) CT_3: Add a new boolean field variable and initialize it to true.
   (b) TT_4: Tag the added variable as a SyncStateVar.
   (c) CT_1: Add a branch for the located block, with its conditional expression set to the added SyncStateVar.
   (d) RF_{23}: Change the body of the else block as required.

Refinement Process

- RF_1: Combine and merge branches with the same result (same behaviour) into a single conditional expression.
- RF_2: Remove clone codes by moving out the same fragment of code repeated in all branches of a conditional expression.
- RF_4: Simplify the conditional by extracting methods from the condition, then part, and else parts.
- RF_{15}: Replace Conditional with polymorphism by moving each branch of the conditional to an overriding method in a subclass. Make the original method abstract.
- RF_{18}: If there is more than one alternative block, then replace nested conditional with Guard Clauses to clarify the default path of execution.
- RF_{19}: This refactoring can be used to convert parameter adaptation provided by an *Select Block* to compositional adaptation using *Replace Block*.

Considerations

- *alternative blocks* must be unreachable in the default application's control flow. This guarantees that in the case that no adaptation is required, the application will preserve its default behaviour.

- In most cases, it is favorable to locate minimal code blocks holding default and alternative behaviour. However, if the code is scattered, we have to do other refactorings to reorder the code (if possible) and exclude code segments that are not part of the behaviour.

AT_3: Add Change State Effector

Requires The state variable is located and available.

Transformation Composition

- **Preparation:** $(RF_9)?(RF_{22})^*(RF_{24})?$
- **Main:** (TT_4)
- **Refinement:** (RF_{14})

Preparation Process

- RF_9: If the state variable is implicitly represented by an expression (e.g., expressions in conditionals), then extract a local variable.
- RF_{22}: Split temp variables that temporarily hold the state to be observed and changed.
- RF_{24}: If the located variable is local then convert it to a field variable.

Main Process

1. TT_4: Tag the Field variable with @SyncStateVar.

Refinement Process

- RF_{14}: If several elements of an array are tagged as SyncStateVar, then replace the array with an object, tag it with SyncStateVar, and remove previous tags on individual elements of the new object.

Considerations None.

AT_4: Add Monitor State Sensor

Requires The state variable is located.

Transformation Composition

- **Preparation:** $(RF_8)?(RF_9)?(RF_{22})^*$
- **Main:** $((RF_{21})?(CT_3)(RF_5))?(TT_5)$
- **Refinement:** (RF_{14})

Preparation Process

- RF_8: If the value of the required state variable is implicitly hold by a section of code, then make the assumption explicit with an assertion. The result of the assertion represents the value of the required state variable.
- RF_9: If the state variable is implicitly represented by an expression (e.g., expressions in conditionals), then extract a local variable.
- RF_{22}: Split temp variables that temporarily hold the state to be observed and changed.

Main Process

1. If the located state variable is not available.
   (a) CT_3: Add a new field variable, and initialize it to the value it should represent.
   (b) Update the value of the added state variable at the located locations in the code.
      i. RF_{21}: If the expressions to compute the state have side effects, try to separate them into two methods, and separate queries from modifiers.
      ii. RF_5: In case the expression has more than one statement, then extract the expression as a new method, and assign the return value to the added field variable.
2. TT_5: Tag the located variable as an @StateVar.

Refinement Process

- RF_{14}: If several elements of array are tagged as an @StateVar, then replace the array with an object, tag it with @StateVar, and remove previous tags on individual elements of the new object.

Considerations

- Initializing the value of the variable in Step 1 of the main transformation process may need additional method calls to retrieve the required state data. As adding a method call is not a refactoring and adds new statements, it is mandatory to make sure that the added expression to compute the value of the state variable has no impact on the application's behaviour (e.g., no variable definition) [129]. This step is basically considered to import the state of the operating environment (e.g., middleware) into the application. For example the code may include function calls to get the state of the JVM, current time, CPU usage, and etc.

After preparing the adaptable software and tagging the required adaptability factors, the next step is to generate its runtime model to be used by GRAF. However, the model generation step is optional, as the model can be generated at load-time initialization (see Subsection 5.8.2). But, as the initial runtime model will only change as the adaptable software evolves, this step can be done off-line for performance improvements.

5.6 Runtime Model Generation

GRAF needs a runtime model that is specific to the adaptable software. The runtime model in GRAF contains the state variables but also a behavioral model. This runtime model can be generated from annotations in the source code of the adaptable software. For each annotated program element, a corresponding representation in the runtime model is created, which can be done automatically upon SAS startup.

The model generator of GRAF supports two ways for setting up the runtime model and essentially creating the runtime model graph: (i) by inspecting Java class files of the adaptable software using reflection, or (ii) by loading an existing runtime model. To support the use of GRAF in a modernization context, we need to create an initial version of the runtime model from compiled class files of the adaptable software by using embedded annotations.

We refer to the runtime model's TGraph representation as the `RuntimeModelGraph`. The `RuntimeModel`, then, is an encapsulating component within GRAF. This TGraph can then be queried and transformed at runtime.

The current implementation of the model generation process is straightforward. Its *input* is the root directory of the adaptable software's compiled Java class files and a file name for saving the runtime model. The *output* of this process is the generated runtime model (TGraph). First, all available classes are loaded from the input directory recursively. For each class, the following steps are performed:

1. Load the class by name using Java reflection.
2. For each field annotated as `@StateVar` or `@SyncStateVar`, create a corresponding `StateVariable` vertex.
3. For each method annotated as `@InterpretationPoint`, create a corresponding `Activity` vertex.
4. For each method annotated as `@Action`, create a corresponding `OpaqueAction` vertex.
5. Setup the activity model with an `InitialNode`, a `FinalNode` and the default action in between to represent the original behaviour.

Optionally, the runtime model can be generated without starting the SAS. The model can be stored persistently in a *TG* file that can be passed to GRAF at startup. Preprocessing steps can be introduced to adjust and refine the initially generated runtime model; for example, the default behaviour representation can be manipulated.

5.7 Preparing GRAF

This section describes how to prepare a customized version of GRAF based on the specified adaptation model.

5.7.1 Implementing Adaptation Rules

The internal adaptation manager of GRAF uses event-condition-action rules that specify what to do given a specific state. In this case, rational behaviour is essentially compiled as

a set of adaptation rules in a form of logical predicates. Currently, adaptation rules need to be programmed in Java. However, GRAF-based SAS can be also managed externally. This allows the GRAF-based system to utilize goal-based and utility-function policies. Based on the chosen external adaptation manager and derived policies, other types of runtime models, such as state charts and petri nets, can be used to model the system's adaptation needs. Based on the adaptation policies and prepared adaptable software, the creation of adaptation rules involves the following steps:

- Step 1: Pick an adaptation requirement.
- Step 2: Choose from a set of identified candidates the state variables that can be used to determine when the adaptation rule shall be applied.
- Step 3: Choose from the set of identified candidates the methods that are marked as interpretation point and that need to be adjusted. In the case of parameter adaptation, this is the getter method for the respective state variable.
- Step 4: Identify the value range of state variables in which adaptation needs to be performed for each of the chosen state variables.
- Step 5: Choose from the set of available alternate actions the ones that can support the adaptation.
- Step 6: Write the concrete adaptation rule for GRAF by:
  - encoding the condition as queries on the state variables, and
  - encoding actions as transformations of the runtime model.

Adaptation rules are implementations of the `IAdaptationRule` interface of GRAF, in the form of *Event-Condition-Action* (ECA) rules. The rule engine is notified of events by the model manager and iterates over the subscribed adaptation rules for the triggered event. A Java template for writing an adaptation rule is illustrated in Listing 5.2.

```
1 public class SampleRule extends AbstractAdaptationRule {
2   public SampleRule() {
3     super();
4
5     // Specify the selection condition.
6     this.condition = new GReQLQueryImpl("Boolean Expression");
7
```

```
8       // Specify the pre-and post-conditions.
9       IQuery preCondition = new GReQLQueryImpl("Boolean Expression");
10      IQuery postCondition = new GReQLQueryImpl("Boolean Expression");
11
12      // Specify the transformation of the TGraph.
13      this.action = new AbstractTransformation(preCondition, postCondition) {
14        @Override
15        public boolean execute() {
16          // Use this helper to get a node to start transformation.
17          QueryHelper qh = ModelManagerImpl.instance().getQueryHelper();
18          // Write some transformation using the generated API.
19          // Return true, if model changed, otherwise false.
20          return false;
21        }
22      };
23    }
24  }
```

Listing 5.2: Template For Writing An Adaptation Rule

The *condition* is a query in a form of a GReQL expression, to be evaluated on the runtime model by the model manager. The *action* is represented by the `ITransformation` interface. GRAF provides an abstract implementation for users of the framework with the `AbstractTransformation` class. The `preCondition` and `postCondition` fields are used by the `AbstractModelManager` for further filtering *before* execution, as well as to make sure that the outcome of the adaptation rule is the expected one *after* the transformation. Some simple transformations are wrapped in the `TransformationHelper`, which can be also obtained from the `ModelManagerImpl` instance. Other transformations can be defined as either GReTL statements or implemented using the Java API, provided by JGraLab.

5.7.2 Advanced Customization

Beside implementing adaptation rules, in most cases there is no need to modify and customize GRAF. However, the extensible and customizable architecture of GRAF allows developers to change and customize GRAF as needed. In this section, we will describe how to extend and customize the runtime model schema, adaptation manager, and middleware adapters of GRAF. We try to avoid implementation details and describe the customization in a more abstract way. However, more details on GRAF implementation and customization is available in [49].

Customizing Runtime Model Schema

The runtime model conforms to a schema (meta-model). The default schema is implemented using the IBM Rational Software Architect (RSA), as UML Class Diagrams. This schema is illustrated in Appendix B. Different views (diagrams) are used to create the full schema. As the current realization of GRAF uses UML Activity Diagrams for modeling behavior at the runtime model level, the runtime model schema must express their *abstract syntax*. Furthermore, state variables are used to store exposed data from the adaptable software. Accordingly, every runtime model is composed of a set of `StateVariable` vertices and by a set of `Activity` vertices. These are the individual UML Activity models to be interpreted at an associated interpretation point.

Generally, there is no need to change and modify the runtime model schema. The schema for these models is available and can be customized as needed. Upon changing the schema, we need to regenerate the API to work with the runtime models to be generated and conformed to the new schema. Based on the runtime model schema, this Java API set is generated automatically using JGraLab tools. The Java classes in the generated API are the types that represent vertex- and edge-types of runtime models (TGraphs) that conform to the runtime model schema. This API can be used to (i) *transform* the runtime model, (ii) *create* types that hold the result of a GReQL query, or (iii) *query* the runtime model in plain Java, without GReQL.

Customizing the Adaptation Manager

The internal adaptation manager and its rule engine can be substituted for more intelligent and powerful adaption managers, such as the one presented in [4]. This goal is achievable in two ways: (i) internally, by extending the available abstract adaptation manager and adaptation rules, or (ii) externally, by connecting external adaptation manager via the provided external management interface. The general interface for adaptation management in GRAF is the `IAdaptationManager` interface, which can be used to implement adaptation managers.

Customizing Adapters

GRAF provides (i) a set of adapters for propagating (*reifying*) values to state variables in the runtime model, and (ii) a generic adapter for interpreting the behavioral model. These adapters are responsible for the proper communication between the adaptable software

and the framework. All middleware adapters implement the provided middleware adapter interface. For propagating values from Java fields (`@StateVar`) to the runtime model as well as for receiving values from it (`@SyncStateVar`), an abstract class is implemented first. The implementation includes basic checks for errors and catches common exceptions that can happen during the synchronization of values (e.g., a corresponding vertex is not found in the runtime model). The provided abstract class for state variable adapters offers two abstract methods, that are called from within the same class, in a form of a *template method*, which is also used as an *AOP advice* during the binding step of instrumentation. When the advice code is invoked, it calls its associated template method first, which has to implement the actual propagation of the value, according to the Java field's type. Every adapter implements this method. Similarly, a generic Interpretation Point adapter is provided by GRAF and its advice code is used in a similar way to state variable adapters.

Listing 5.3 contains a sample implementation of the template method for integer data types. Using a helper method also provided by the abstract base class, the Java API that was generated from the `runtime-model-schema` project is used to get a reference to the associated `IntegerType` vertex in the runtime model.

```
1  @Override
2  protected Object toRuntimeModelAdviceBody(FieldWriteInvocation invocation)
    throws Throwable {
3    // Get the correct node from the model by id.
4    StateVariable sv = getStateVariableForInvocation(invocation);
5
6      // Add any customizations for data modification, filtration, conversion,
         etc.
7      // ...
8    // Now the data is ready to be passed to the runtime model.
9
10     // Change the value in the runtime model.
11   ((IntegerType) sv.get_type())
12       .set_value((Integer) invocation.getValue());
13
14   return invocation.invokeNext();
15 }
```

Listing 5.3: Implemented Template Method In `Integerstatevariableadapter`

5.8 Integration and Deployment

This process is responsible for merging and integrating the three system elements (adaptable software, Adaptation Rule Repository, Model Constraint Repository) that are prepared by the previous steps of the modernization process. The Integration process can be split into the startup configuration and the load-time initialization phases.

5.8.1 Startup Configuration

When using GRAF, users of the framework need to configure the startup of their application. The configuration consists of two steps: (i) initializing an instance of GRAF, and (ii) initializing an instance of the instrumented application.

In the first step, we need the paths of:

- The adaptable software (source or binary),
- Adaptation rules, and
- Constraints.

In the second step, there are two parameters that must be passed to glue the application to the framework:

- The path to the JBossAOP library.
- The path to an XML file that configures the *AOP* pointcuts for reification and interpretation.

5.8.2 Load-Time Initialization

Starting the GRAF as described in Subsection 5.8.1 invokes a chain of initialization steps that are automatically performed behind the scenes. When an instance of GRAF is created, the following steps need to be executed to make sure the framework can operate as expected. The GRAF implementation performs them automatically and users of the framework do not need to manually configure them. Yet, in cases where an extension or configuration of GRAF becomes necessary, developers should be aware of the initialization procedure. The required steps are as follows:

- Set up all of the GRAF components:
  - Generate the runtime model based on the annotated source code.
  - Create and link the model interpreter to the created runtime model.
  - Create the default rule engine.
  - Register the rule engine as an observer of the rule engine via the model manager.
- Add adaptation rules to the rule repository. Sample code behind this step is sketched in Listing 5.4.

```
1 AdaptationRule ar = new LoadBalanceRule();
2 GRAFFactory.instance().getAdaptationManagementLayer().getRuleRepository
  ().add(ar);
```

Listing 5.4: Adaptation Rule Setup Snippet

- Add model constraints to the constraint repository. Sample code behind this step is sketched in Listing 5.5.

```
1 ModelConstraint mc = new NoDublicatesConstraint();
2 GRAFFactory.instance().getRuntimeModelLayer().getConstraintRepository()
  .add(mc);
```

Listing 5.5: Runtime Model Constraint Setup Snippet

Certain system elements of GRAF perform lazy loading, such as the repositories. Objects will be created when first needed. Furthermore, each layer is implemented as a singleton, providing access to only those internal components which are meant to be used by other layers.

5.9 Summary

In this chapter, we proposed a modernization process model for a systematic evolution of the original software towards model-centric self-adaptive software. The process is developed on top of the conceptual model proposed in Chapter 3 and the developed model

centric adaptation framework, GRAF. The process starts by specifying adaptation and adaptability problem spaces, and modeling the adaptation and adaptability separately. These two models served as basis for two separate paths to prepare adaptable software, and the adaptation manager. The process is iterative, which means as we progress through the process we update and adjust specifications and design models as required. Changes in adaptation and adaptability spaces may demand a change in the other space, which shapes a co-evolutionary problem to build adaptation manager and adaptable software as two artifacts that are interacting with each other via an abstract connection domain, the runtime model.

The architecture of GRAF exhibits a loosely coupled integration between adaptation manager and adaptable software based on the proposed sensing and effecting styles. It help us to minimize the required evolution changes to be made on software to a set of adaptability transformations to include missing adaptability factors in software. These transformations can be defined as a sequence of refactoring changes that preserve the original behaviour of software when there is no need for adaptation. This is extremely important for a modernization process, since one of the major common drawbacks of moving towards self-adaptive systems was the complexity and error-proneness of the required changes to make software adaptable.

After describing the modernization process, an assessment is required to determine the applicability of our proposed approach on real world cases and applications. That is the goal of the next chapter, applying the proposed approach on a set of case studies as a proof of concept.

Chapter 6

Case Studies

"*Inside every large program, there is a small program trying to get out.*"
–C.A.R. Hoare

Case studies are an essential research methodology for applied disciplines [54]. Regardless of how a case study is used, be it for theory building or theory testing, it is a process of scholarly inquiry and exploration whose underlying purpose is to create new knowledge. As a proof of concept, and in order to put the approach of this thesis into practice, we performed a number of case studies in order to address the following research questions:

- **RQ1:** What is the impact of different preparation techniques for adaptable software (for the same adaptation requirements) on the performance of SAS?
- **RQ2:** What are the benefits and drawbacks of a GRAF-based SAS system?
- **RQ3:** Is the proposed modernization process capable of evolving software for self-adaptation in a cost-effective manner?

The case studies were conducted on two real-world Java-based software systems, belonging to two different application domains that can greatly benefit from self-adaptation: Internet telephony, and computer games. Table 6.1 summarizes the properties of the selected software for both case studies.

The first case study aims to answer **RQ1** by demonstrating the importance of planning. Throughout this case study we exemplify how various versions of an adaptable software can be prepared for a fixed set of requirements, demonstrating the effectiveness of the proposed

Table 6.1: Some Statistics on the Selected Case Studies

System	Domain	Version	License	Language	LOC
OpenJSIP [143]	Telephony	0.0.4	GNU GPL	Java	6K
Jake2 [91]	Game Engine	0.9.5	GNU GPL	Java	126K

model-centric approach, and GRAF's ability to support both parameter and compositional adaptation in a simple scenario within the voice-over-IP (VoIP) domain.

The second case study is mainly designed for answering **RQ2** by describing the more complex scenario of making a legacy game engine self-adaptive. The main focus of this case study is on analyzing the cost and overhead of adaptation via using and managing models at runtime in the GRAF approach.

Moreover, both of the conducted case studies serve to demonstrate the applicability and cost-effectiveness of the modernization process in answer to **RQ3**. Throughout the case studies, we follow the modernization steps taken for preparing adaptive version of both systems.

For each case study we present: (i) the design of the case studies, (ii) the system setups, and (iii) the experiments and their results. However, before describing the case studies, we give an overview of the measures we used for evaluating the quality and cost-effectiveness of our approach. In addition, the case studies are followed by two discussion sections targeting the research questions and the threats to validity of the case studies.

6.1 Measures

The exact measurement of cost and effectiveness in software systems is one of the major software engineering challenge, which is beyond the scope of this thesis. However, in order to address the research questions of our case studies, and addressing the objectives of this thesis we use a simplification of the general forms to measure the cost and effectiveness of adaptation and evolution changes. The rest of this section, will describe the measures used in our case studies, and the rationale for selecting them and a discussion on their ability to measure the variables under study.

6.1.1 Evolution Cost

Evolution cost is combination of the effort to perform the change and the effort to maintain it. In GRAF approach evolution changes are divided into 1. preparing the adaptable software via adaptability transformations, as described in Section 5.5, and 2. preparing the customized version of GRAF. In order to measure the required effort for change, we use product metrics. Among product metrics, software size is thought to be reflective of complexity, development effort and reliability of software, which is commonly used by many well-known cost estimation and prediction models (e.g., Basic COCOMO and COCOMO II) [17]. Moreover, complexity is directly related to software reliability, so representing complexity is also important. Thus, we measure the McCabe's Cyclomatic Complexity metric for measuring the complexity of added or modified methods. Table 6.2 lists these product metrics.

Table 6.2: List of the Measured Product Metrics

Metric	**Description**
Cyclomatic Complexity (CC) [124]	counts the number of code conditions giving an indication of how complex the program is.
Executable Statements (EXEC)	This metric counts the number of executable statements.
Non-Comment Lines of Code (NCLOC)	This metric counts all the lines that do not contain comments or blank lines.

Along with the product metrics, we define and measure a set of metrics that represent different aspect of software adaptivity. Table 6.3 lists the defined metrics to measure and provides a brief description for each metrics. Using these metrics to measure software cost is similar to using *function points* for measuring the effort and cost of software changes [17].

On top of the measured metrics, test coverage metrics are a way of estimating fault and reliability by performing tests on software products, based on the assumption that software reliability is an indicator of the required future changes.

6.1.2 Evolution Effectiveness

In order to measure the effectiveness of the modernization process, we need to measure if the required adaptability is provided by the evolved software (i.e., adaptable software). Software is considered adaptable if and only if it exposes the required monitoring data via its sensor interface, and exposes the required effecting operations via its effector interface.

Table 6.3: List of the Measured Adaptivity Metrics

Metric	**Description**
Number of Adaptation Rules (NAR)	The number of adaptation rules.
Number of Annotations (NAN)	Number of exposed adaptability factors via annotations.
Number of Effectors (NOE)	The number of exposed effectors.
Number of Sensors (NOS)	The number of exposed sensors.
Number of Queries (NQ)	Total number of GreQL queries for all adaptation rules.
Number of Transformations (NT)	Total number of transformations (GreTL & JGraLab) for all adaptation rules.
Software Adaptability Degree (SAD)	Measure the required software adaptability by counting the number of adaptation goals that their required sensors and effectors are provided by the software system.

Similar to other non-functional properties, adaptability is not a binary property and any software has some degrees of adaptability. To have synergy with other quality properties and their corresponding non-functional metrics, we define a new quality metric, *Software Adaptability Degree (SAD)*, to measure software adaptability. SAD is an indicator of the application's adaptability regarding the given adaptability specifications. To be more precise, SAD can be defined as follow:

- Let $AS(S, E)$ be an *Adaptable Software* where $S \neq \emptyset$ is the set of sensors exposed through a sensor interface, and $E \neq \emptyset$ is the set of effectors exposed through an effector interface.
- Let $G_k(Y_k, U_k)$ be an *Adaptation Goal*, where Y_k is a set of sensors required by G_k to detect a symptom, and U_k is a set of possible effectors to be triggered by a selected action in G_k.
- Let $AM(Y, U)$ be an *Adaptation Manager* to control the set of adaptation goals $\{G_1, \ldots, G_k\}$, where $Y = \bigcup_{k=1}^{z} Y_k$ is the set of all required sensors by a sensor interface, and $U = \bigcup_{k=1}^{z} U_k$ is the set of all exposed effectors through an effector interface.
- Let $SAD(x)$ be the Adaptability Degree of software $x(S, E)$, where $SAD(x) = |G|$ and $G = \{\forall_i G_i(Y_i, U_i) | Y_i \subset S \wedge U_i \subset E\}$.

Given $AS(S, E)$ and $AM(Y, U)$, $SAD(AS) = \max(SAD)$ if and only if $Y \subset S$ and $U \subset E$. SAD is a top level software adaptability metric for measuring the effectiveness of evolution processes to make software adaptable. In other words, SAD measures to what extend the prepared software satisfies observability and controllability requirements demanded by the adaptation manager.

6.1.3 Adaptation Cost

In this case study, we measure the following two important factors as an indicator of adaptation overhead, proposed by Salehei [160]:

- Resource overhead of the adaptation mechanism.
- Mean-Time-To-Adapt (MTTA) as an indicator of mechanism performance.

In the conducted case studies, our measure for adaptation cost is set to monitoring the resource usage and resource allocation of the systems under test and method level profiling to compute the execution times.

6.1.4 Adaptation Effectiveness

Evaluating the effectiveness of the adaptation process is measured as a function of where takes into account the number of times the adaptation goals have been denied or satisfied, or how much goals were deviated from their activation criteria [160]. However, in the absence of explicit goals, for attribute-action coupling mechanisms, we can use utility values for evaluating the adaptation effectiveness.

Therefore, measuring adaptation effectiveness is completely case dependant based on the defined adaptation goals. In our case studies, we defined a utility function for evaluating the system reliability. However, the effectiveness of the adaptation in the second case study (user satisfaction) is completely user dependant, and cannot be measured qualitatively.

6.2 OpenJSIP: A Stand-Alone Telephony Server

Several researchers have mentioned telecommunication as a promising domain for the application of an SAS (e.g., [145]). For the first case study, we selected *OpenJSIP* [143], a

stand-alone open-source Java SIP server for voice over IP calls. OpenJSIP provides three distributed services: *Proxy*, *Registrar*, and *Location Service*, and is based on JainSIP [90], the standardized Java interface to the *Session Initiation Protocol* (SIP) for desktop and server applications.

Throughout this section, we present the steps performed to evolve OpenJSIP towards runtime adaptivity using GRAF. We also compare the behaviour of the target system to the original system and demonstrate the ability of the evolved software to fulfill a given adaptation requirement.

6.2.1 The Adaptation Requirement

Software reliability is defined as [121] "the probability of failure-free software operation for a specified period of time in a specified environment". Reliability is one of the most important *Quality of Service* (QoS) properties for enterprise systems such as telephony. Poor reliability can strongly affect the user's satisfaction when the system shows unexpected behaviour that in turn can have a negative impact on its business value. Thus, we set our adaptation requirement for this case study as:

- *AR*: *The system shall adapt its behaviour to maintain its reliability under extreme system loads.*

We simplify the adaptation effectiveness measurement (i.e., maintaining reliability) by assuming that resource saturation under extreme loads is the only root of failure in OpenJSIP; this means that no failures due to bugs or underlying layers such as the physical, network, and operating system layers are taken into account. Regarding this assumption, OpenJSIP either succeeds or fails in handling calls, so we define reliability as the percentage of successfully handled calls.

$$Successfully\ Handled\ Calls = Serviced\ Calls + Rejected\ Calls \tag{6.1}$$

and we define system's *Failure Rate* as the percentage of failures within the test period, divided by the total time a function of successfully handled calls:

$$Failure\ Rate = 100 * (1 - \frac{Serviced\ Calls + Rejected\ Calls}{Total\ Calls}) \tag{6.2}$$

Note that in under this definition, rejecting a call is not the best action and shall be selected only when necessary. It is the role of the adaptation manager to analyze the situation and to choose the most appropriate action.

6.2.2 Adaptation Requirements Analysis

Here our adaptation goal is to increase the reliability, which we will measure by the defined utility function, as defined in Equation 6.2. In order to satisfy AR, we decided to intentionally reject (not service) the calls, when the system load is high, and switch back to the normal service mode when the system load is reduced. Intentionally rejecting the calls will prevent system overload, which can result in abnormal system behaviour and/or request timeouts. Therefore, we will need a *domain attribute* to sense the system load, and an *adaptation action* to reject the calls. This can be shown in a form of the following two action-rule policies:

- P_1: `IF` *load is high* `THEN` *reject new registrations.*
- P_2: `IF` *load is low* `THEN` *service new registrations.*

Our goal in this case study is to follow to separate paths for implementing adaptivity for satisfying P_1 and P_2 at the application-level: (i) *compositional adaptation*: adding or removing a filter method, which rejects calls that originate from new registrars, or (ii) *parameter adaptation*: performing a static structural change by adding a conditional statement in the source code and dynamically controlling the application's flow by externally tuning the decision criteria at this turning point.

6.2.3 System Analysis

We analyzed OpenJSIP and identified the required adaptability factors in its source code and mapped them to their supporting program elements. We comprehended and investigated the source code manually. The adaptability factors locate the pointcuts to be instrumented and to be exposed to GRAF. The list of potentially needed adaptability factors for parameter as well as compositional adaptation is presented in Table 6.4, together with their availability status in the source code.

This step is similar to classic concept location and impact analysis procedure, which is extensively addressed in the domain of software evolution and maintenance. We perform the concept location manually. However, concept location tools and techniques (e.g., JRipples [31]) and [70]) may be used instead to facilitate this step.

Table 6.4: List of Adaptability Factors for Compositional and Parameter Adaptation of OpenJSIP

ID	Type	Status	Description
AF_1	StateVar	located	Monitor the server load.
AF_2	Action(Alternative)	missing	Reject new registrations.
AF_3	Action(Default)	located	Allow new registrations.
AF_4	InterpretationPoint	missing	Redirect flow of control.
AF_5	SyncStateVar	missing	Switch between AF_2 and AF_3.

6.2.4 Planning for Change

We need AF_1 for reifying the server load in both approaches to achieving adaptivity. We have two possible options for implementing adaptivity: providing AF_4 for compositional adaptation, or AF_5 for parameter adaptation.

In the first case, AF_2 is an additional action that rejects incoming register messages to the proxy, and AF_3 performs the connection for incoming registration requests, which is the implemented default behaviour. AF_4 is the definition of the interpretation point in OpenJSIP's control flow, at which register events shall be either accepted or rejected. To cover the second approach, AF_5 is a synchronized state variable that needs to be used as the condition to toggle between accepting or blocking the register events.

6.2.5 Preparing Adaptable OpenJSIP

Independent of the technique used for effecting the system at runtime, we need to provide the system's load. Necessary modifications to the class `Proxy` are shown in Listing 6.1. We use the number of current server transaction and define a state variable named `currentSrvTrans` that represents AF_1. This variable is kept up to date using the added helper method `updateCurrentSrvTrans()`.

```
1 public class Proxy {
2   @StateVar
3   private static int currentSrvTrans;
4
5   public void updatecurrentSrvTrans() {
6     BigInteger tempSrvTrans = (BigInteger) snmpAssistant.getSnmpInteger(
      SNMP_OID_NUM_SERVER_TRANSACTIONS).getValue();
```

```
7       currentSrvTrans = tempSrvTrans.intValue();
8     }
9   }
```

Listing 6.1: Exposing the System Load, Measured in Current Server Transactions

Scenario I: Parameter Adaptation

To achieve parameter adaptation, AF_5 is used at a turning point to tune its decision criteria, so as to switch between the default behaviour (AF_3) and the new alternative action (AF_4). Listing 6.2 shows a snapshot of this change. In order to preserve the default behaviour of OpenJSIP in the no-adaptation case, we initialize the value of `blockRequests` to `false`.

```
1   public class Proxy {
2     @SyncStateVar
3     private static boolean blockRequests = false;
4
5     private void processIncomingRequest(final RequestEvent requestEvent) {
6       // blockRequests value is set from model.
7       if (method.equals(Request.REGISTER) && blockRequests) {
8         // P2: Reject message and return.
9       }
10        else {
11      // P3: Otherwise process messages.
12        }
13    }
14  }
```

Listing 6.2: Using a Tunable Synchronized State Variable

The value of `blockRequests` is controlled by the rule engine using an adaptation rule graphically shown in Figure 6.1. The adaptation rule makes use of types, defined at the runtime model schema level. State variables from the runtime model's structural model are resolved by name. The condition of the rule depends on the current state of blocking as well as on the amount of server transactions. The latter value depends on the system that executes the proxy, as well as on the saturation point of OpenJSIP for handling transactions.

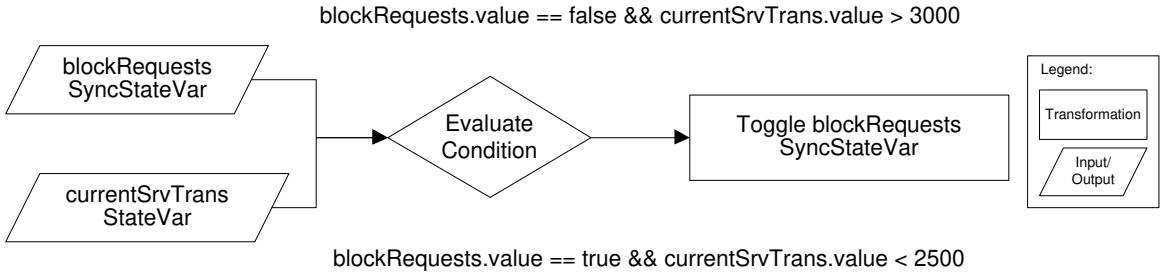

Figure 6.1: Adaptation Rule for Parameter Adaptation Scenario

Scenario II: Compositional Adaptation

Another possibility for implementing adaptivity is by separating behaviour in the form of two separate actions, one for accepting (AF_2) and one for rejecting calls (AF_3). Depending on the orchestration of actions as defined in the behaviour description (UML activity) for the interpretation method, messages are accepted or blocked. The necessary code template is shown in Listing 6.3.

```
1  public class Proxy {
2    @InterpretationPoint
3    private void processIncomingRequest(RequestEvent requestEvent) {
4      acceptRegisterRequest(requestEvent);
5    }
6
7    @Action(interpretationPointKey = "openjsip.proxy.Proxy.
       processIncomingRequest")
8    private void processIncomingRequestBody(final RequestEvent requestEvent) {
9      // Accepting registration message.
10   }
11
12   @Action()
13   private void dropRequest(final RequestEvent requestEvent) {
14     // Rejecting registration message.
15   }
16 }
```

Listing 6.3: Setup of an Interpretation Point and Two Actions

If the runtime model is not transformed, the body of `processIncomingRequest()` is executed. Otherwise, the model interpreter executes the associated behaviour in the model.

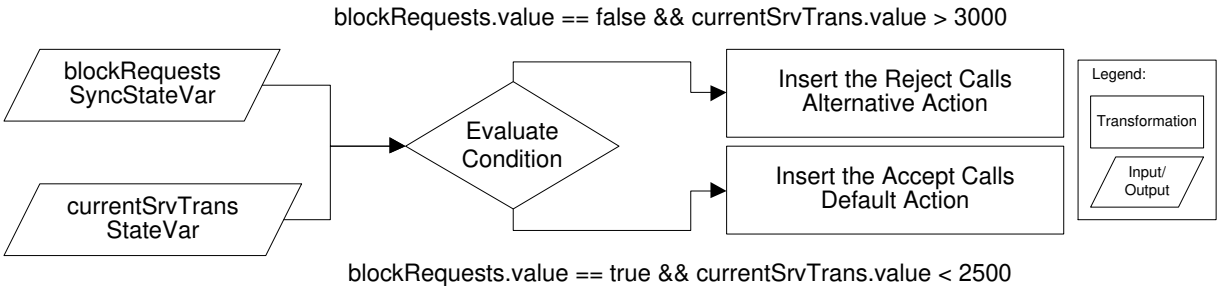

Figure 6.2: Adaptation Rule for Compositional Adaptation Scenario

The call to the interpreter is hidden from the developer. The code is generic and injected using AOP. Figure 6.2 shows the corresponding adaptation rule.

The measured product metrics for the original and two prepared adaptable versions of OpenJSip are presented in Table 6.5. Here, both adaptable versions include a new added method, `updateCurrentSrvTrans()`, to get current load. This method is called in appropriate place in the `processRequest()` method to update the added state variable, using the *Add Monitor State Sensor* adaptability transformation. For the parameter adaptation the `processIncomingRequest()` is changed to include the alternative flow (branch) by applying the *Add Select Block Effector* transformation, while for the compositional adaptation, the body of `processIncomingRequest()` is extracted as `processIncomingRequestBody()` and new alternative methods are added accordingly, following the *Add Replace Block Effector* transformation.

Table 6.6 presents the measured adaptability metrics for the two prepared adaptable versions of OpenJSIP. As both versions fully address the required adaptation goals their SAD value is 1. Here, the main difference is in the number of annotations (NAN) for both versions, as the compositional adaptation version due to the separation of default and alternative actions has more adaptable elements.

6.2.6 Results

We tested both of the evolved versions of OpenJSIP and the original system in a simulated environment. Our goal was to study the applicability of our approach and to evaluate the performance of the adaptive systems in various conditions. To achieve this goal, we set up a local network with two workstations, one dedicated to a server, and one for two SIP traffic generators. We generated client traffic using SIPp [171]. All systems were tested under identical conditions.

Table 6.5: Measured Method-Level Metrics for OpenJSIP

System	Method	CC	NCLOC	EXEC
Original	main(String[])	8	59	15
Parameter	main(String[])	8	72	17
Compositional	main(String[])	8	72	17
Original	processIncomingRequest(RequestEvent)	13	280	3
Parameter	processIncomingRequest(RequestEvent)	14	289	3
Compositional	processIncomingRequest(RequestEvent)	1	1	0
Original	processRequest(RequestEvent)	2	17	2
Parameter	processRequest(RequestEvent)	2	22	2
Compositional	processRequest(RequestEvent)	2	22	2
Compositional	updateCurrentSrvTrans()	1	3	1
Parameter	updateCurrentSrvTrans()	1	3	1
Compositional	dropRequest(RequestEvent)	1	5	0
Compositional	processIncomingRequestBody(RequestEvent)	13	280	3

Table 6.6: Measured Adaptivity Metrics for OpenJSIP

System	NAR	NAN	NOE	NOS	NQ	NT	SAD
Parameter Adaptation	2	2	1	1	6	2	1
Compositional Adaptation	2	5	1	1	6	2	1

The main testing scenario was as follows: we generated unique callers and callees at various calling rates. All clients try to register their ID's at the server (callees first). Then, each caller tries to call a predefined callee by sending an `INVITE` message to the server. The server looks for the callee and if the callee is available, it informs the caller to establish the call. The caller holds the line for 20 seconds and then tries to terminate the call by sending a `BYE` message to the server. Both sides terminate the call when they receive an `ACK` from the server. The clients keep generating traffic at different rates, i.e, calls per second (CPS), for a total time of 5 minutes.

Prior to starting the stress tests, we had to detect the saturation point of OpenJSIP on the computers used. We tested the original system with an increasing traffic and observed 18CPS as the limit with around 3000 server transactions. This threshold was used to adjust the adaptation rules' conditions.

Table 6.7: OpenJSIP Stress Testing Results
N.: Normal, P.A.: Parameter Adaptation, C.A.: Compositional Adaptation

CPS	**System**	**Reg. Attempt**		**% Suc. Reg.**		**% Suc. Call**		
		AVG	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	
6	Original	3593.53	4.27	100.00	0.00	100.00	0.00	
	Parameter	3593.80	3.86	100.00	0.00	99.89	0.01	
	Method	3598.53	4.82	100.00	0.00	99.94	0.01	
20	Original	11981.00	0.00	88.31	0.00	85.56	0.00	
	Parameter	11983.67	5.51	89.41	3.79	91.59	4.02	
	Compositional	11980.67	15.04	85.64	7.01	90.30	6.62	
40	Original	23968.33	37.53	59.43	1.04	56.53	1.04	
	Parameter	24021.67	46.37	49.74	9.09	78.52	3.73	
	Compositional	23967.67	22.01	47.13	6.95	75.77	2.99	
Dyn	Original	13786.33	2.31	74.45	1.09	76.68	0.64	
	Parameter	13786.00	7.94	69.13	10.98	87.44	11.49	
	Compositional	13798.33	35.23	68.70	8.57	86.63	14.80	

CPS	**System**	**% Suc. End**		**%Handled**		**Failure Rate**		**%Gain**
		AVG	**STDEV**	**AVG**	**STDEV**	**AVG**	**STDEV**	
6	Original	99.94	0.01	100.00	0.00	0.00	0.00	0.00
	Parameter	100.00	0.00	99.97	0.01	0.02	0.00	-0.02
	Compositional	100.00	0.00	99.97	0.01	0.02	0.00	-0.02
20	Original	96.69	0.00	79.08	1.15	19.45	0.62	0.00
	Parameter	96.51	1.38	88.04	4.13	10.38	2.98	9.07
	Compositional	93.26	2.80	81.63	8.10	16.79	6.29	2.65
40	Original	70.25	0.71	41.81	1.45	56.60	0.83	0.00
	Parameter	92.06	11.40	83.79	8.55	15.97	8.70	40.64
	Compositional	91.02	6.08	79.20	3.08	20.53	3.53	36.07
Dyn	Original	87.20	1.34	61.55	2.32	37.51	0.20	0.00
	Parameter	96.39	7.64	86.15	10.10	13.53	9.61	23.98
	Compositional	94.72	1.69	84.43	7.66	15.23	7.16	22.28

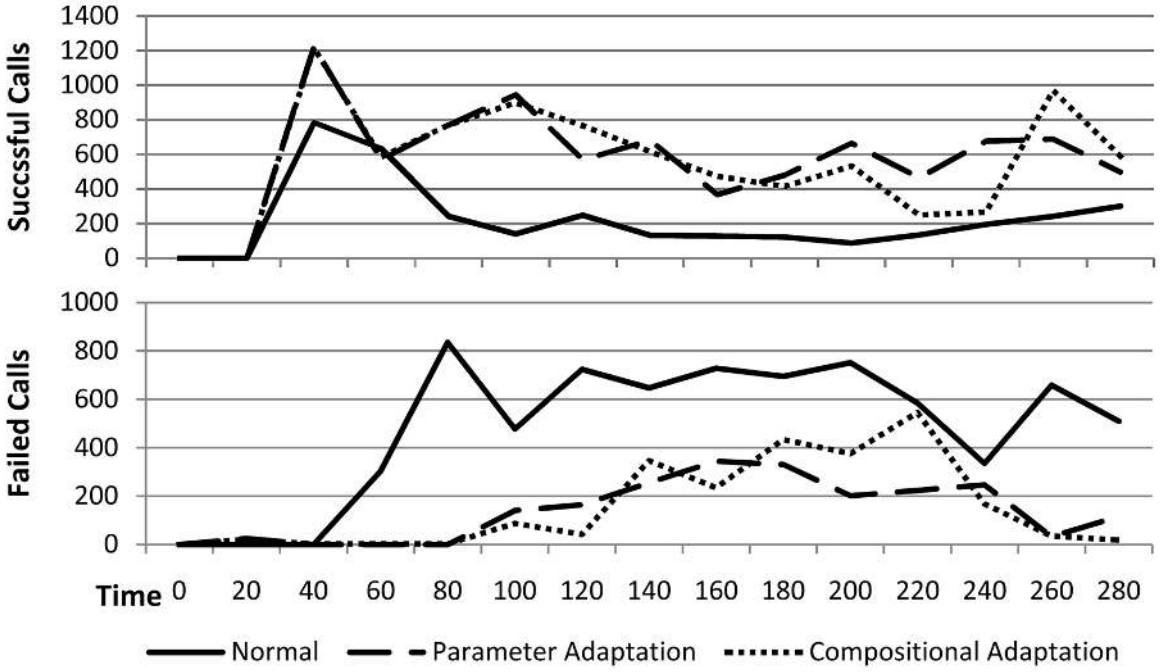

Figure 6.3: Comparison Among the Rate of Successful and Failed Calls

We tested the adaptive systems at four different rates of traffic: (i) 6CPS for normal load traffic, which is 1/3 of the original system's boundary, (ii) 20CPS for high load traffic, which is just above the threshold level, (iii) 40CPS for extreme load, and (iv) a dynamic load sequence of 10, 20, 40, 30, and 15CPS for one minute each.

The results are shown in Table 6.7. They reveal that the adaptive behaviour improves the system's ability to handle more messages successfully. The first row in Table 6.7 shows the system's performance at 6CPS. The adaptive approaches perform identical to the original system. OpenJSIP did not reach its saturation point and all established calls are ended successfully with a `BYE` message (Suc. End).

The heavier the load gets, the more effective the adaptive behaviour becomes. At 20CPS, around the point of high load, we can see minor improvements in the lightweight parameter adaptation approach, but the number of handled messages for the compositional adaptation drop below that of the original application. We attribute this drop to the added overhead for transforming and interpreting the activity.

The advantage of adaptation becomes clear in the case of the extreme load of 40CPS. A more detailed overview of the performance gain at this extreme load is presented in

Figure 6.3. Although our approach of blocking new incoming registrations under high load is a simple one, OpenJSIP is now able to handle nearly twice the amount of messages compared to its non-adaptive version. Again, parameter adaptation performs slightly better than compositional adaptation in our implementation.

6.3 Jake2: A Legacy Game Engine

Providing the best quality of service and performance is crucial in modern games [152], and gamers are always eager to be challenged by more intelligent computer opponents. We believe that computer games can extensively benefit from autonomic computing and runtime adaptation. Thus, we selected the Jake2 legacy game engine as our second case study to demonstrate how to use GRAF for enabling model-centric adaptation. Jake2 [91] is a Java port of the GPL release of the Quake II game engine, a classic first person shooter game that was originally written in C and released in 1997.

6.3.1 The Adaptation Requirement

In Jake2, a player is able to select the game's difficulty level as the game starts. Naturally, players improve their performance as they spend more time playing the game. However, the game's difficulty level is fixed unless the player restarts the game, and because the system is not getting any feedback regarding the player's performance, the game might become too easy or too hard for the player. Hence, we set our main adaptation requirement as follows:

- AR': *As the game progresses, the computer-controlled artificially intelligent enemies (bots) should adapt their behaviour according to the performance of the human player.*

6.3.2 Adaptation Requirements Analysis

This requirement can be achieved by altering the way bots play the game; thus, we need to change the behavioural model of bots at runtime. Four candidate alternative behaviours to satisfy AR' are:

- *Approaching Enemy Differently:* use different path algorithms for approaching the enemy. For example, instead of running straight towards the target position, the bot

could start strafing left and right to be harder to hit by a player. This adaptation can be used to make the game harder for good players.

- *Adaptive Weapon Selection:* instead of a fixed weapon, the bot can adaptively select the most appropriate weapon. For example, set the attack state randomly to either missile or melee, or always use a melee weapon, even when far away to reduce the game difficulty.
- *Cloning and Morphing:* the bot creates clones of itself. We can consider a number of clones, and cool-down times. Clones can be initialized by setting their current and max health, weapons, and ammo.
- *Morphing:* the bot re-spawns itself as a stronger/weaker bot. New bots states can be also initialized by a set of provided state variables, such as current and max health, weapons, and ammo.

Each behaviour is adjustable by tunable parameters. Due to the similarities in specification, modeling, and implementation of the alternate behaviours, this study focuses on two selected adaptive behaviours: *cloning* and *morphing* as listed in Table 6.8

Table 6.8: List of Alternative Bot behaviours to Adjust Jake2's Game Experience

behaviour	**Parameters**	**Description**
Cloning	*numberOfClones*, *cooldown*	The bot creates clones of itself.
Morphing	*botID*, *botState*	The bot re-spawns itself as a stronger/weaker bot.

Given the list of alternative behaviours, we specify AR' by the following adaptation policies:

- P_1: `IF` *the player's performance is bad* `THEN` *stick to default attack.*
- P_2: `IF` *the player's performance is good* and *default behaviour is in place* `THEN` *enable cloning* xor *enable morphing.*
- P_3: `IF` *the player's performance is supreme* `THEN` *enable morphing and cloning.*

Here, the parameters to indicate bad, good, or supreme performance are adjustable. The performance of the player changes as the game difficulty varies or as time elapses.

6.3.3 Preparing Adaptable Jake2

We explored the original source code of Jake2 and located the elements that implement the bot behaviour, based on the state variables that carry information about the performance of human players. Top level modules (packages) of Jake2 are listed in Table 6.9. Furthermore, we studied the `jake2.game` package, which implements all game logic (triggers, bots, weapons, elevators, explosions, secrets, etc.). The `jake2.game.GameAI` class contains a set of methods that implement the intelligent behaviour of bots. We investigated its methods to determine how the bots perceive, decide, and react in their world.

Table 6.9: Main Components of Jake2 [91]

Package	Description and Content
client	Contains all visual effects, key handling, screen drawing, message parsing, etc.
common	common support functionality (e.g., filesystem, config variables).
game	all game logic (e.g., triggers, bots, weapons, elevators, explosions, secrets).
install	the jake2 installer.
renderer	3D graphics displays.
server	world handling, movement, physics, game control, communication.
sound	sound implementations and a sound driver.
sys	methods dealing with the operating system (e.g., network, keyboard handling).
util	3d maths, generic utility methods.

In Jake2, every bot is always standing or running. As a bot perceives a target, its state changes from *idle* to *attack*. This transition is implemented by calling `GameAI.ai_checkattack()`. We used UML activity diagrams during the program comprehension step, because this representation is close to the behavioural model representation in GRAF (see Figure 4.5).

A bot runs towards his next target without firing and checks if the target is dead or not. If it is dead, the bot tries to find a new target. Otherwise, it will show its hostile animation and update its own knowledge about the distance to the target. If the target is still available and visible, the bot tries to attack in various ways.

In first person shooter games, a basic indicator of a player's performance is the rate of killing enemy bots. We decided to monitor the amount of killed monsters per game session. This data is stored in the `level_locals_t` class, a structure that is cleared when each map is entered, and is read/written to a save-file for persistency. The `killed_monsters`

integer field of this class is the only state variable and represents the total number of killed monsters. We annotated this field as `@StateVar`.

Furthermore, we needed to change `GameAI.ai_checkattack()` in order to modify the behaviour of bots, so we exposed this method by annotating it as `@InterpretationPoint`. In GRAF, each action in the behavioural model maps to a unique method in code. Therefore, we refactored the original `ai_checkattack()` method and extracted a set of new actions from its body. Because the required actions to perform morphing and cloning were missing, we added two alternative actions to `GameAI`, namely `ai_tryClone()` and `ai_tryMorph()`. We annotated each of the extracted methods and the two new ones as `@Action`.

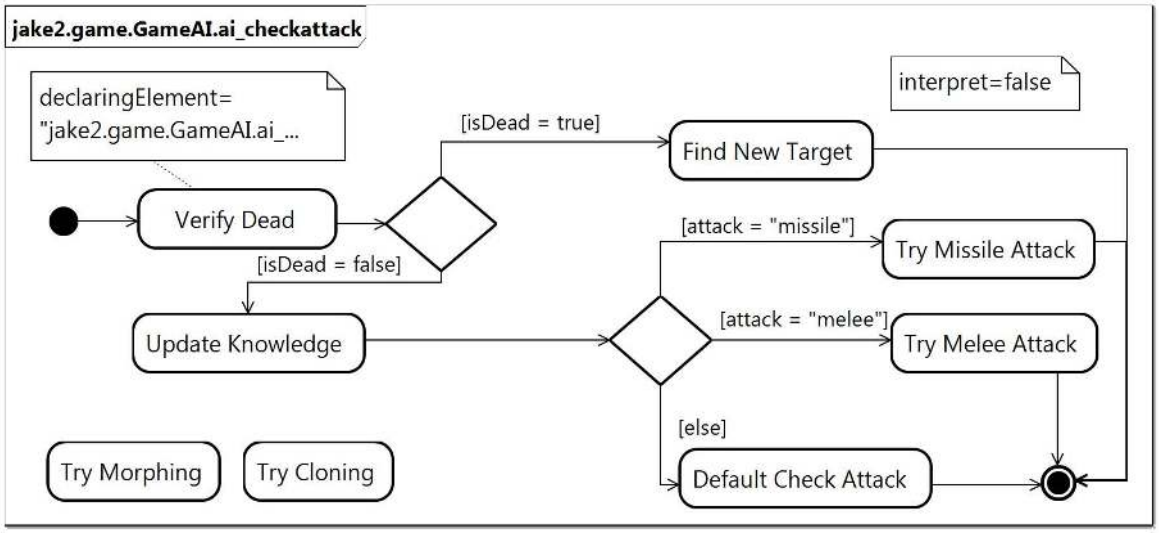

Figure 6.4: UML Activity Diagram Representing the Bot's Default behaviour

Figure 6.4 illustrates an excerpt of the runtime model, which represents the default attack behaviour for bots. We use the concrete syntax of UML activity diagrams here. The internal representation of the actual TGraph derived from this model conforms to the runtime model schema. The activity's name matches the name of the method that is annotated as `@InterpretationPoint`. Furthermore, it is set to be *not* interpreted, as noted in Figure 6.4 by a UML comment, because UML activities have no properties.

Finally, the two floating actions *Try Cloning* and *Try Morphing* represent the alternative actions that can be used within this behaviour description. The second annotation in this figure denotes the `declaringElement` attribute of the `Action` vertex, which is used

by the model interpreter to invoke an actual method of the adaptable software. This meta-data is only shown for one action here.

6.3.4 Developing Adaptation Rules

In this section, we discuss, by example, the adaptation rule that is responsible for cloning. Figure 6.5 illustrates the outcome of the transformation and shows the changes to the default behaviour.

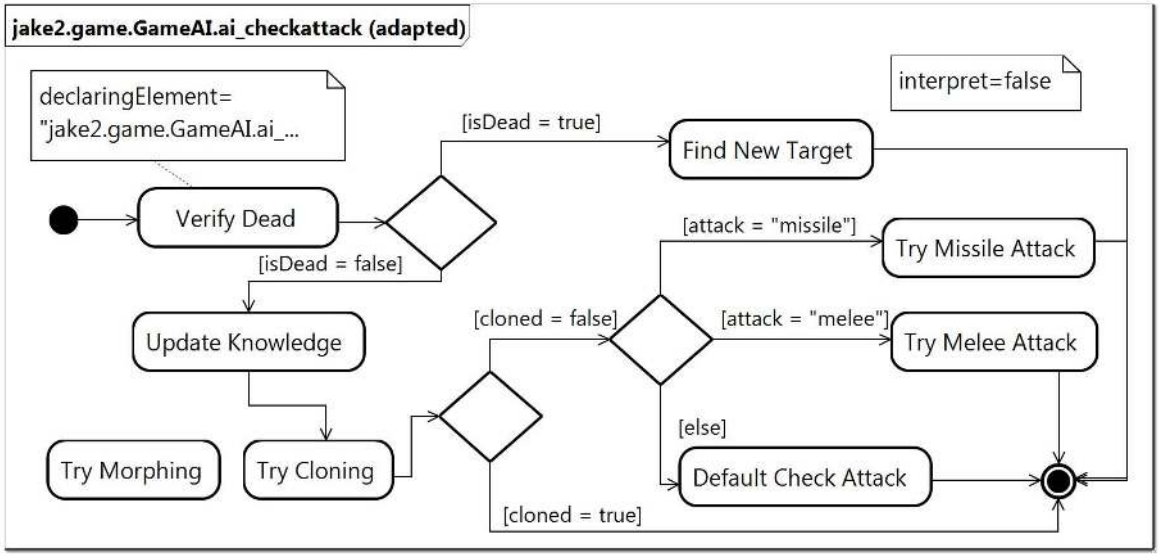

Figure 6.5: UML Activity Diagram Representing the Bot's Adapted behaviour for Cloning

The transformation redirects the outgoing edge of *Update Knowledge* to point to *Try Cloning*, so that it does not connect to the second `DecisionNode` vertex any more. A new edge is created to connect *Try Cloning* to a new, third `DecisionNode` vertex, where the output of *Try Cloning* is checked using a simple GReQL expression. These expressions can make use of the names of `OutputPin` vertices (shown in the schema, but hidden in this example view). For the case where the cloning was successful, the third decision node is connected to the `FinalNode` vertex. Otherwise, bots should check for a missile or melee attack as before.

Listing 6.4 illustrates an extract of the adaptation rule with a condition to check the condition for evaluating player's performance. Here the condition will be evaluated to true

if the player has killed more than 10 bots. Although, the selected rule is a simple one and may not fully reflect player's in game performance, the rationale to present this rule is to show a simple GreQL query and how it is implemented as a part of the adaptation rule.

```
1  public class CloningStartRule extends AbstractAdaptationRule {
2
3      private static final String ADAPTED_behaviour_NAME = "jake2.game.GameAI.
       ai_checkattack";
4      private static final String TIME_TO_CLONE_VAR = "jake2.game.
       level_locals_t.killed_monsters";
5
6      public CloningStartRule() {
7          super();
8
9          // Select this rule after 10 bots are killed
10         this.condition = new GReQLQueryImpl(
11                 "exists killed : V{state.StateVariable}, "
12                         + "killType : V{state.LiteralInteger} @ "
13                         + "killed -->{state.HasValue} killType and "
14                         + "killed.name = \"" + TIME_TO_CLONE_VAR
15                         + "\" and killType.value > 10");
16     // ...
17     }
18 }
```

Listing 6.4: Sample GreQL query to check the condition for evaluating player's score

Listing 6.5 illustrates an extract of the adaptation rule's action to add cloning of enemies. After defining the name of the transformation and the needed schema elements are imported in line 3. This transformation works in-place and is structured as `MatchReplace pattern <== modelElements`. `modelElements` is a set of runtime model elements selected using a GReQL query. The pattern is then executed on the result of this query, which selects vertices and edges (`From`), defines restrictions on their relationships and attribute values (`With`) and returns the result (`reportSet`). Following a graph grammar-like approach, the pattern defines how to match and replace elements in this result set to achieve the described adaptation of the activity model.

The measured product metrics for the original and adaptable version of Jake2 is presented in Table 6.10. The methods added to the original version of Jake are the result of the decomposing the original `checkattack()` method into its composing actions, by applying the *Add Replace Block Effector* transformation. Here, the only included alternative

```
transformation CloningStartTransformation;

import behaviour.*;

MatchReplace

('$.ms') <--{'$.fr1'} ('$.f1') -->{'$.t1'}
('$.tc'),
('theElement($.tc <>--{HasOutputPin})' | name = '"cloneOutPin"')
<--{behaviour.From} (behaviour.Flow) -->{behaviour.To}
dn(behaviour.DecisionNode | name = '"Return OR Try Missile"')
<--{behaviour.From} (behaviour.Flow | guardCondition = '"using cloneOutPin:
 cloneOutPin = true"') -->{behaviour.To}
('theElement(V{FinalNode})'),
dn <--{behaviour.From} (behaviour.Flow | guardCondition = '"using
 cloneOutPin: cloneOutPin = false"') -->{behaviour.To} ('$.tm')

<==

from
  middleStuff, tryMissile, tryClone: V{OpaqueAction},
  missileOutPin: V{OutputPin}, decNode: V{DecisionNode},
  flow1: V{Flow},
  from1: theElement(edgesFrom{From}(flow1)),
  to1: theElement(edgesFrom{To}(flow1))
with
  tryClone.name = "ai_tryClone"
  and middleStuff.name = 'ai_middleStuff'
  and tryMissile.name = 'ai_tryMissile'
  and tryMissile <>--{HasOutputPin} missileOutPin
  and middleStuff <-from1- flow1 -to1-> tryMissile
  and missileOutPin <--{From} & {Flow} -->{To} decNode
reportSet
  rec(ms : middleStuff, tm : tryMissile, tc: tryClone, f1 : flow1, fr1 :
   from1, t1 : to1)
end;
```

Listing 6.5: Excerpt of GReTL Transformation in Adaptation Rule for Cloning Functionality

action is cloning, which is consists of a main method, `doClone()`, and a helper method `tryClone()`. Moreover, Table 6.11 presents the measured adaptability metrics for the prepared adaptable version of Jake.

Table 6.10: Measured Method-Level Metrics for Jake

System	Method	CC	NCLOC	EXEC
Original	jake2.game.GameAI.ai_checkattack(edict_t, float)	23	77	30
Adaptable	jake2.game.GameAI.ai_checkattack(edict_t, float)	7	13	1
Adaptable	jake2.game.GameAI.ai_checkIfAtCombatPoint(edict_t, float)	8	20	5
Adaptable	jake2.game.GameAI.ai_checkIfDead(edict_t, float)	8	19	1
Adaptable	jake2.game.GameAI.ai_checkVisibility(edict_t, float)	1	1	0
Adaptable	jake2.game.GameAI.ai_defaultCheckAttack(edict_t, float)	1	1	0
Adaptable	jake2.game.GameAI.ai_findNewTarget(edict_t, float)	4	16	8
Adaptable	jake2.game.GameAI.ai_middleStuff(edict_t, float)	2	13	9
Adaptable	jake2.game.GameAI.ai_tryMelee(edict_t, float)	2	5	1
Adaptable	jake2.game.GameAI.ai_tryMissile(edict_t, float)	2	5	1
Cloning	jake2.game.GameAI.ai_doClone(edict_t, float)	2	22	14
Cloning	jake2.game.GameAI.ai_tryClone(edict_t, float)	5	14	4

Table 6.11: Measured Adaptivity Metrics for Jake

System	NAR	NAN	NOE	NOS	NQ	NT	SAD
Adaptive Jake	4	12	1	1	6	2	1

6.3.5 Evaluating GRAF/Jake2 Performance

To investigate the usability of GRAF for enabling runtime adaptivity in Jake2, we prepared five system variations:

- S_0: The original version of Jake2.
- S_1: the refactored and annotated version of S_0 as described in Subsection 6.3.3.

- S_2: S_1 integrated with GRAF. Aspect codes are woven at annotations. There are no adaptation rules, which means that the default behaviour is always in effect. The default behaviour is achieved by executing the original source code.
- S_3: Similar to S_2, but the default behaviour is achieved by interpreting the default runtime model and skipping the original implementation.
- S_4: Adaptation rules and model constraints are added to S_3. GRAF transforms and interprets the bot behaviour.

Comparing S_1 (original version) to the various adaptive versions of Jake2 allow us to measure the overhead of the refactorings for preparing the actions, aspect weaving, runtime model, model interpretation, and adaptation rules. Table 6.12 summarizes the difference among of the all variants.

Table 6.12: Main Characteristics of Jake2 Evaluated Variations

ID / Characteristic	S_0	S_1	S_2	S_3	S_4
Actions & Annotations	-	x	x	x	x
Start GRAF & Runtime Model	-	-	x	x	x
Interpretation & Reflection	-	-	-	x	x
Adaptation Rules	-	-	-	-	x

All of the experiments were run on an Intel Core 2 Duo E8400 workstation with 4.0GB DDR2 memory, running MS Windows XP Professional 32-bit SP3 and Java Platform, Standard Edition 6. We used JConsole (from Oracle JDK 6), the Eclipse Test and Performance Tools Platform 4.7.1 [187], and Fraps 3.2.6 [66] for the purpose of profiling and benchmarking Jake2 variations.

In executing each variation, we waited for one minute for the game's start menu to load, and started a new game with the default settings and with beginner difficulty. Following a fixed path, we played the game for 2 more minutes before returning back to the spawn location in the game.

Memory Utilization

Investigating the utilized resources among different variations of Jake2 can help us to study the usability and scalability of GRAF. Table 6.13 summarizes the memory utilization of

each variation and the total number of classes loaded. The results for S_0 and S_1 are almost identical, which indicates that we can safely replace the original software with its adaptable version from this perspective.

Table 6.13: Summery of Memory Benchmark for Jake2 Variations

Metric \ ID	S_0	S_1	S_2	S_3	S_4
Max Used (MB)	98.2	98.8	170.5	165.9	171.0
Max Committed	112.4	112.4	213.6	213.1	215.2
Loaded Classes	3799	3799	5648	5660	5754

As we integrated and started GRAF in S_2, S_3, and S_4, we observed a constant increase in memory utilization. There was an average of 72% increase in the total amount of used memory (heap and non-heap), 30% increase in the maximum committed memory, and 50% increase in the total number of loaded classes.

In conclusion, in the case of using Jake2 with GRAF, we had a quite significant, overhead in memory use as we moved from non-adaptive variations to the self-adaptive ones. However, model interpretation and the adaptation rules did not add a significant memory overhead. Loading the JGraLab libraries as well as the Java representation of the runtime model are the main causes for this increase in memory usage.

Execution Performance

All variations exhibited a similar overall game performance: 46 ± 1 average CPU% and 67 ± 1 frames per second. This means that the memory overhead of using GRAF and runtime adaptation did not affect the overall gaming experience.

We further investigate the execution performance of the `GameAI` class and specifically `ai_checkattack()` as summarized in the box-plot of Figure 6.6. The evaluation results reveal that the adaptable Jake2 variant (S_1) containing the refactored `ai_checkattack()` method has an almost identical performance as compared to S_0. S_2 has a slight overhead upon GRAF startup, and when the annotated elements are wrapped by aspect code. The execution times are higher, but still within a acceptable range, in S_3 and S_4.

In these two adaptive versions, the desired behaviour of the bots was achieved via interpretation. S_4 has higher upper quartile than S_3, because the alternative behaviour model in S_4 also performs bot cloning. Additionally, there are more upper overlays in S_4

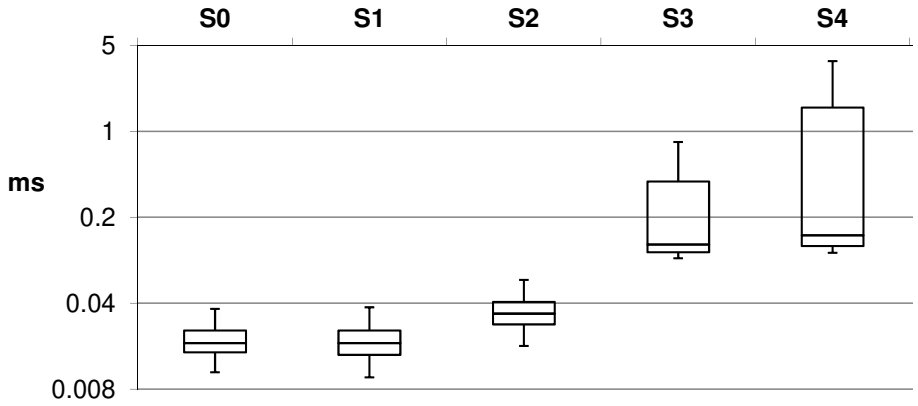

Figure 6.6: Box-Plot of Execution Times for `GameAI.ai_checkattack()`

Table 6.14: Impact of Weaving Aspects on First Calls of `GameAI.ai_checkattack()`.

	S_0	S_1	S_2	S_3	S_4
First Call Execution Time (ms)	1.5	1.4	94.8	172.9	141.6

because the adaptation rule is enabled and occasionally transforms the runtime model. Another observation is the overhead of dynamically weaving the aspect code, as shown in Table 6.14, which only affects the first call to each annotated program element. However, this can be prohibited by weaving the aspect code offline.

6.4 Discussions on the Obtained Results

The following sections provide the relevant answers for each of the research questions.

6.4.1 The Importance of Having Proper Adaptable Software

The hypothesis associated with **RQ1** is: *(H1) Given a specific adaptation requirement, the prepared adaptable software plays an important role in the performance of the final SAS.* To test this hypothesis, we performed the OpenJSIP case study, in which we prepared two different adaptable versions of the original application.

The results demonstrate that both implementations provide the requested adaptive behaviour of rejecting new registrations. Both scenarios do not change the system's behaviour

at normal levels of load and are even able to significantly improve the performance under high loads.

Compositional adaptation is the more flexible approach for implementing adaptivity, because the system's behaviour can be controlled by the runtime model's behavioural model; however, this approach adds some overhead. In contrast, parameter adaptation seems more suitable for cases that do not require highly dynamic structural change at runtime. Parameter adaptation has less overhead on system performance, mainly due to the fact that part of the structural change is done during the evolution process at the source code level, and no explicit model interpretation is needed.

Our behaviour model is kept simple in the OpenJSIP case study, but more complex behaviour including sequences of actions and decision points could be used to describe complex behaviour based on composition of Adaptive Actions.

6.4.2 The Advantages and Disadvantages of GRAF-based SAS

The hypothesis associated with **RQ2** is: *(H2) the model-centric architecture of GRAF supports developing and maintaining complex SAS systems.* The second case study, Jake, was mainly designed to validate this hypothesis. Fulfilling runtime adaptation was more complex than in the first case study, which allowed us to evaluate our approach more extensively.

An advantage of the adaptation framework is that it has a clear structure and its components are well defined in terms of their responsibilities and tasks. Additionally, integrating GRAF with adaptable software is just a matter of marking program elements with one of the provided annotation types. The instrumentation and the binding is transparent to developers. Furthermore, an initial version of the runtime model is generated automatically, which reduces the amount of manual modeling work to be done, and additionally ensures that naming conventions are followed.

On the down side, developing adaptation rules is not easy, because this task requires knowledge about the actions provided by the adaptable software. Furthermore, the developer must be familiar with GReQL and GReTL in this implementation of GRAF. Not surprisingly, the lack of proper debugging facilities showed up as a fundamental problem. During the development of adaptation rules, adequate visualizations of the runtime model, as well as illustration techniques for maintaining the relationships and interplay between adaptation rules and their effect on the runtime model was missing.

Finally, it must be noted that the clear separation of concerns between the ways of

(i) creating software that is made adaptable by modifying existing software, and (ii) developing adaptation rules and constraints with minor knowledge about the adaptable software's internals, is a clear advantage. Major disadvantages resulted from the lack of tailored tool support and are not directly related to GRAF's design.

6.4.3 The Cost-Effectiveness of the Approach.

The hypothesis associated with **RQ3** is: *(H3) the proposed model-centric approach, and its supporting modernization process are capable of evolving software for self-adaptation in a cost-effective manner.* To test this hypothesis, we modernize the original source code of both case studies by following the steps of our modernization process.

We successfully evolve both case studies towards GRAF-based SAS, and to evaluate cost-effectiveness, we demonstrate reuse and ease-of-use of adaptability transformations with GRAF. We show that following the modernization framework in conjunction with GRAF provides common and reusable infrastructures, which are flexible to customize to make a system self-adaptive. In effect, GRAF saves engineers time and development effort to add and evolve self-adaptation capabilities to a target system. To show effort savings, we demonstrate how developers can prepare adaptable software by applying adaptability transformations and preparing GRAF by including the required adaptation rules. As adaptability transformations are basically composed from a set of classic refactoring, preparing adaptable software can be automated for further improvement of the approach cost-effectiveness. Having refactorings instead of arbitrary evolution changes reduce the cost of change and improves the quality of software [65].

In order to check the future maintenance cost of the prepared applications, we measured a set of product metrics, as well as the proposed adaptivity metrics and the results show that the added complexity, and size is not significant compared to the project size.

6.5 Threats to Validity

As we discuss GRAF's qualities based on experiences gathered from case studies, it is important to describe the possible threats to validity as well as ways to prevent their occurrence. Several factors could influence the results of the two studies and hence, could affect our inferences. Therefore, we cover internal and external validity.

6.5.1 Internal Validity

The threats in this regard are mono-operation and mono-method biases. Regarding the results of the performance measurements, the used/developed tools for collecting data can potentially have defects or affect the performance results themselves. Efforts to restrict this threat were mitigated by considering various profiling tools, applying them to several case studies, and studying their impact on the results. In addition, only widely used software was used, such as JConsole, a JMX-compliant monitoring tool that is shipped with the official Java development kit.

For Jake2, we tried to reproduce the exact game scenario for each experiment. However, this was not fully achieved due to the game's random nature. One solution to face this threat would be to use customized test maps as well as a scripted player that follows a pre-defined path automatically. Moreover, eliminating any randomness could be achieved by using pseudo-random generators with fixed seeds. However, this threat does not apply to OpenJSIP, as the calls are deterministically generated using the SIPp traffic generator.

Neither of the case studies were performed in an embedded and real-time environment, which may effect the measured performance results. Furthermore, the JVM provides dynamic memory management and garbage collection, which can affect the software's performance as well. For instance, the captured memory utilization results may be influenced. We manually triggered the garbage collection and used averaging to tackle this threat. We preferred a more realistic environment over an isolated and rather artificial one, but still, we minimized the number of background processes when running each case study.

Another known internal threat is our measure for evaluating the cost of evolution changes. In general, measuring the evolution cost is challenging and itself is a costly process, which constitutes measurement of the effort required to fulfill all the sub-processes of the modernization process model (see Figure 5.2). However, in this research the conducted case studies were especially selected, analyzed, and designed towards the goals of our case studies. These case studies were also used for the purpose of testing and debugging the implementation of GRAF. That was the main reason to select and measure product metrics, which are widely used in software cost-estimation models [17]. This simplification can be considered as an internal threat to the validity of our results. In order to reduce or eliminate this threat, we need to conduct a set of empirical analysis on a number of case studies selected from different domains, and ask candidate developers (with or without domain and system knowledge) to modernize the selected systems towards GRAF-based SAS, and measuring the exact amount of time spend to fully modernize each system under study and make it operational.

6.5.2 External Validity

Common threats to external validity are related to mono-operation and mono-method biases. We demonstrate the results of two case studies here to reduce this threat, but more experience is needed to be able to widely generalize our results.

We believe that the selected scenarios are detailed enough to show a reasonably complex behavioural model. However, each case study is limited to a single adaptation requirement, and more complex situations with possibly contradictory adaptation rules were not covered. Nonetheless, we considered several candidate solutions to achieve the adaptation goal for each case study. In addition, each solution can be realized differently. Two candidate scenarios, parameter and compositional adaptation, were presented for OpenJSIP.

Other legacy software with different characteristics, e.g., mission and safety critical systems, may be subject to different adaptation concerns, and the overhead of integrating and using GRAF may differ. Controlling this threat can be achieved by applying additional empirical studies using various software projects, preferably from different domains and on a set of different hardware systems.

6.6 Summary

The goal of this chapter was to put the proposed approach and GRAF into action. The main objective was to check the usability and cost effectiveness of the approach on a set of real-world case studies. For each case study we used the proposed modernization process to make self-adaptive version of the original systems. We run each evolved software and evaluate its ability and effectiveness to satisfy given adaptation requirements. The results of the case studies supports in favor of the applicability and effectiveness of the thesis approach. There also exists a number of threats to the validity of the case studies, and we tried to highlight them and give a solution to reduce or eliminate them as a part of the discussion section.

It is noteworthy that although we captured and presented some detailed performance results in our case studies, benchmarking was not the main objective here. Therefore, instead of performing syntectical analysis in isolated test environments, we tried to simulate more realistic test environments for both case studies.

It is also notable to mention that the initial challenge in these experiments was the lack of a commonly used testbed or benchmark, and a considerable amount of time was spent on selecting case studies and preparing the platform to apply the adaptation. This part

of the work has been accomplished collaboratively with Mahdi Derakhshanmanesh from Institute for Software Technology at the University of Koblenz-Landau and Greg O'Grady from the Software Technologies Applied Research Laboratory (Star Lab) at the University of Waterloo.

Chapter 7

Concluding Remarks

"*Adapt or perish, now as ever, is nature's inexorable imperative.*"
H. G. Wells

In order to effectively engineer and use self-adaptive software systems, in this thesis we proposed a new conceptual model for identifying and specifying problem spaces in the context of self-adaptive software systems. Based on the foundations of this conceptual model, we utilized the concept of model-centric adaptation, and proposed a novel model-centric approach for engineering self-adaptive software by designing the architecture of a generic adaptation framework and an evolution process to support it all together. This approach is tailored to facilitate and simplify the process of evolving and adapting current (legacy) software towards runtime adaptivity, and the conducted empirical studies reveals the applicability and effectiveness of this approach to bring self-adaptive behaviour into non-adaptive applications that essentially demand adaptive behaviour to survive. This thesis addresses the main research challenges towards its objectives as follow:

- *Specifying Adaptation Requirements:* As discussed throughout the thesis, the self-adaptive software problem domain can be divided into the two main problems of adaptation and adaptability. Given a set of adaptation requirements, specifications need to be defined for the adaptable software (i.e., adaptability specifications) and the adaptation manager (i.e., adaptation specifications). The conceptual model and detailed metamodels presented assist requirements engineers in understanding the SAS domain better, and in specifying each of these subsystems based on the shared phenomena of their associated domains. Although we do not consider this conceptual model as an architecture

model for addressing the solution-space concerns of self-adaptive software, due to its tight connection with existing software phenomena, it can also serve as a design model. For example, compared to the IBM autonomic computing reference architecture [98], this model is more generic. In the autonomic computing model, the interfaces between the adaptation manager and its application domain do not exist. In other words, the adaptation manager only has shared phenomena with the adaptable software, but does not share any phenomena with its domain. Hence, in the case of open-loop control, the sensor and effector interfaces should be exclusively specified based on the shared phenomena between the adaptation manager and the adaptable software.

- *Managing the Complexity of Self-Adaptive Software:* To tackle the complexity issue in the context of SAS systems, the construction of precise and accurate models of software that address the demands of self-adaptive systems is a necessary but hard task, given their highly complex and dynamic nature. Difficulties arise especially in the area of model-centric runtime adaptation, where models need to be generated, manipulated, and managed at runtime. In theory, model-centric approaches are able to reduce the complexity of engineering and maintaining an SAS, by providing partial and customized representations of the software, tailored to suit the needs of adaptation. However, in practice, creating, managing, verifying, reflecting, and maintaining the consistency of runtime models adds additional complexity and overhead to the system, which hinders the usability and cost-benefit ratio of model-centric approaches to self-adaptive software. This phenomenon conforms to Lehman's second law of software evolution: "as software system is changed, its complexity increases and become more difficult to evolve unless work is done to maintain or reduce complexity" [112]. Using adaptation frameworks that are specifically tailored to contain and manage models at runtime can partially reduce the complexity of using models at runtime, and enhance the efficiency of model-centric approaches. Consequently, the proposed concept of having adaptability models for specifying runtime models, using an extensible metamodel, can greatly facilitate the program comprehension step of migrating legacy systems towards runtime adaptivity.

- *Preparing Self-Adaptive Software:* In this thesis, we gave an overview of how our graph-based runtime adaptation framework (GRAF) supports model-centric runtime adaptivity based on a combination of querying, transforming, and interpreting the behavioral models and state variables that are causally connected to a software application. Another strength of GRAF that distinguishes it from other adaptation frameworks is its ability to support method-level compositional adaptation by interpreting runtime models that are not necessarily at the same abstraction level and expressiveness of the base-layer that they represent. By minimizing the required changes to make software adaptable

without breaking the default functionality, GRAF reduces the risk of moving towards self-adaptive software systems. Consequently, the proposed concept of having adaptability models to specify runtime models, with a flexible metamodel, can greatly facilitate the program comprehension and concept location step of migrating legacy systems towards runtime adaptivity.

- *Planning for Software Changes:* From our perspective, the use of models at runtime is a natural extension of using models at design or compile time. Note however, that the requirements for models at each phase of these *binding times* are different, for example, with respect to performance requirements. For instance, model sizes and processing times for queries and transformations become especially important at runtime, particularly when applied under real-time constraints. The above perspective combines classic software maintenance and evolution with dynamic evolution for the purpose of adaptation. Hence, the modernization process (as in classic software maintenance) for creating an adaptable software is not separate from run-time adaptation. Having such a unified view of the complete engineering process was one of the main motivations of this research.

In order to evaluate the proposed approach for evolving software systems towards self-adaptation, a set of empirical studies were conducted on two real-world scenarios, from two diverse domains that can greatly benefit from self-adaptation. The main objectives of the conducted case studies are to examine the applicability and effectiveness of the proposed modernization process, the performance and usability of GRAF, and the tradeoffs among different change plans for a unique set of adaptation requirements. The following conclusions can be drawn from the case studies:

- Adaptability transformations can produce adaptable software from the original software.
- The separation of concerns between the ways of (i) creating software that is made adaptable by modifying existing software, and (ii) developing adaptation rules and constraints with limited knowledge about the adaptable software's internals, is a clear advantage.
- Given a specific adaptation requirement, the prepared adaptable software plays an important role in the quality of the final SAS system.
- There is a performance overhead associated with using GRAF to perform model-centric runtime adaption. However, the major part of this overhead does not increase with the size of the adaptable software or the complexity of the required adaptation.

- Activity models can accurately and effectively represent the behavioural model of the software at runtime, and their interpretation can be done almost as fast as the execution of their matching binary code.

- Compositional adaptation is more flexible than parameter adaptation for the purpose of implementing adaptivity, because the system's behavior can be controlled by the runtime model's behavioral model; however, this approach adds some overhead. In contrast, parameter adaptation seems more suitable for cases that do not require highly dynamic structural changes at runtime. Parameter adaptation has less overhead on system performance.

This research is a step towards establishing a cost-effective process for (re)engineering self-adaptive software systems. Some assumptions and limitations were taken to narrow the scope of this thesis, and the boundaries of the proposed approach. The most important ones are the following:

- The proposed conceptual model is developed based on control-based self-adaptive software, which is the most common approach for self-adaptation.

- We assume that adaptation requirements are provided as the starting point for engineering SAS. However, a complete engineering cycle starts with the elicitation or derivation of adaptation requirements from stakeholders or system artifacts.

- Using GRAF, the adaptation changes are checked against set of model-level and application-level constraints and invariants. However, these checks do not necessarily cover all existing invariants and constraints.

- The modernization process and GRAF are designed for object-oriented Java applications. They also assume that the application's source code is accessible and modifiable. However, this assumption is not valid for many legacy systems.

- Formulating adaptation impacts for the purpose of verification and validation is currently realized as a set of queries on the runtime model. However, this procedure can be done by static/dynamic analysis, and through some rigorous tests of the software. This task is particularly significant in engineering SAS, but is out of the scope of this thesis.

7.1 Future Research Directions

Self-adaptive software has a long way to go to be mature and trustable, and many challenges apply to the theoretical and practical aspects of engineering these systems. We see a number of interesting directions which could improve the engineering process to achieve self-adaptation. Based on the contributions of this thesis and critical review of current research state of the art, here we name few notable topics together with ideas and visions for future research directions:

Verification and Validation

One interesting future direction is to support model validation. The validation of transformed models can be done by getting feedback from the adaptable software and its operating environment, after the completion of the adaptation cycle. A cycle is complete when the transformed runtime model is reflected, and the application behavior has changed accordingly. The results of this behavior alternation can again be captured by the runtime model during the feedback loop. Trends can be calculated using utility-functions or metrics for computing distances from the current state to the desired ones, as formulated by adaptation goals.

Self-Organizing Software Systems

As the trend of modern software is moving towards large scale distributed systems such as cloud computing, we believe that this work, and model-centric adaptation in general, can serve as a foundation for self-organizing systems that can be used in such environments. An external manager can serve as an adapter to connect multiple GRAF-based systems to build a network of self-adaptive software systems. Such a network can be used to create self-organizing systems. Each individual software can share its runtime model, as a clear and consistent representation of itself and its context, with other software. Several designs and algorithms can be developed to efficiently handle a large set of runtime models. These models may also contradict with each other; resolving these conflicts in large scale software domains is an interesting problem to solve. General awareness of the possible actions (as transformations) that each software is able to do, and deciding to choose the best set of actions (a sequence of transformations on disparate software) is another challenging research direction.

Measuring and Benchmarking (Self-)Adaptive Software Systems

There is a need for a metrics suite that is able to provide a better understanding of software adaptability that serve as a base for more comprehensive assessment of software adaptability such as: (i) availability of sensors and effectors according to self-* properties requirements (each self-* property may require a specific set of sensors and effectors), (ii) quality of sensors, and (iii) efficiency of effectors. In this thesis, we proposed *Software Adaptability Degree (SAD)*, a top level metric to measure software adaptability, and more detailed metrics can be also defined based on the sensors and effectors, their composing adaptability factors, and elements that are supporting them in software. Specially, in case of migrating current software systems towards adaptability, these metrics can be used to measure the effectiveness of evolution processes by representing the extent of which the application satisfies the adaptation requirements. More detailed metrics can be also defined based on the sensors and effectors, their composing adaptability factors, and elements that are supporting them in software, especially, in case of migrating current software systems towards adaptability.

Also, there is a need of well-defined benchmarks and case studies in the domain of self-adaptive software systems. Quality benchmarks and measures can greatly improve the ongoing research on these systems and can be used for a fair comparison of adaptation frameworks, techniques, and tools.

Perspective of Intelligent and Adaptive Adaptation Managers

In [4], we show how machine learning can be used in decision making of SAS. This approach is proposed to tackle the action selection problem with the aid of reinforcement learning algorithms. This work demonstrates reinforcement learning as a good technique to tackle the problem of action selection in SAS, and can be extended to an actual implementation to be integrated with GRAF. More experiments with other benchmarks could be performed and the technique could be compared to other works from literature. In terms of improving the performance of the model there are also many opportunities. One could use other methods of reinforcement learning such as Monte Carlo or Q-Learning [180]. Moreover, the parameters of the algorithm could be adjusted, and better mechanisms of multi-objective action selection could be examined.

Identifying design patterns for self-adaptive software

Several architecture styles, design principles, and coding rules have been developed for adaptive and autonomic systems. These design principles are usually presented as architecture or design patterns [73, 154]. On the other hand, many design patterns from relevant disciplines are applicable, such as distributed computing [32] and fault tolerant software [81]. Putting all these patterns together, investigating more patterns, and presenting them as a unified pattern language is an interesting and valued future research direction in the area of engineering self-adaptive systems.

The initial step to develop a comprehensive pattern language for engineering self-adaptive software systems can be shaped based on the the proposed conceptual model of self-adaptive software. Such a pattern language will be similar to the language developed by Buschmann *et al.* [32] in POSA (Pattern-Oriented Software Architecture) for architectural patterns in distributed computing domain, in which they defined a domain model as the root of their language. However, a more diverse set of architectural styles and design patterns is needed to serve as a guideline for engineering self-adaptive software systems.

Design and Development of a Toolchain

Tools to supporting engineering self-adaptive software can divided into (i) *development* tools, and (ii) *maintenance* tools. In terms of preparing software to make it adaptable, tools are needed for viewing those parts of the implementation at a granularity that is sufficient, e.g at the method signature level. Such a view can be integrated into software development tools, and annotations for marking software elements (to expose the four interfaces to GRAF) can be added via context menus. Furthermore, the adaptability transformations can be (semi-)automated using available refactoring tools, script languages, and development environments.

Moreover, queries and transformations are at the heart of this technology. Embedding queries (GReQL) into the Java code is not optimal and writing transformation rules (both GreTL and JGraLab rules) is tedious at the moment. We envision two possible solutions at this point. First, we can develop a domain specific language for writing the most common transformation tasks (together with helper methods for GReQL queries). Second, we can try to offer a fully model-centric approach, where developers see a visual model, edit it and record their steps in terms of add/change/remove operations on elements as described by the runtime model schema. The recorded sequence is the normalized (maybe manually in the beginning) and is transformed to an implementation of the model transformation

in Java. An exchange format is desirable in this case. For instance, we need to provide a simple way for developers (users of GRAF) to create and modify adaptation rules. One way is to implement extensions to popular software development environments (e.g., Eclipse), so that wizards help with the creation of boilerplate code. Syntax highlighting and code completion is also helpful.

Various Behavioral Model Types

The current implementation of GRAF uses a subset of UML activity diagrams for modeling the behavior of software routines. This decision decision suits our objectives, as we mainly target application-level fine-grain adaptation by orchestrating methods rather than coarse-grain adaptation (e.g, component and service composition) at higher abstraction levels. Nevertheless, we are sure that other forms of modeling behavior can be useful for achieving adaptivity for software. Possibly candidates can be petri nets, state charts, or feature models. In all cases, a replacement or even just an adjustment of the runtime model (that is, its schema) involves the adjustment of the model interpreter as well, as the runtime model schema describes the *syntax*, and the model interpreter encodes a description of *semantics*.

Native Code Interoperability

Developed based on Java technologies, GRAF inevitably has stronger support for modern object-oriented applications. To introduce the benefit of adaptivity and modernization process to true legacy systems, we need to experiment with binding the framework to software that was developed using languages that do not rely on a virtual machine. Prominent examples are C code, as well as COBOL. Using the *Java Native Interface* (JNI), we can attempt to play with non-adaptive software and make it adaptive using GRAF.

In this scenario, we will need to change the implementation of the interfaces as well as the way we do the binding of the framework to the adaptable software once it is prepared. In cases where no *Aspect-Oriented Programming* (AOP) is supported, code generation may be considered. Moreover, analyzing the impact of coupling a managed software system to native code in terms of its execution/runtime performance and memory allocation characteristics needs to be investigated, as the feasibility of such an approach will be most likely questioned by system programmers.

APPENDICES

Appendix A

Catalogue of Supporting Refactorings

RF_1**:** Consolidate Conditional Expression

Problem: You have a sequence of conditional tests with the same result.

Solution: Combine them into a single conditional expression and extract it.

RF_2**:** Consolidate Duplicate Conditional Fragments

Problem: The same fragment of code is in all branches of a conditional expression.

Solution: Move it outside of the expression.

RF_3**:** Encapsulate Field

Problem: There is a public field.

Solution: Make it private and provide accessors.

RF_4**:** Decompose Conditional

Problem: You have a complicated conditional statement.

Solution: Extract methods from the condition, then part, and else parts.

RF_5**:** Extract Method

Problem: You have a code fragment that can be grouped together.

Solution: Turn the fragment into a method whose name explains the purpose of the method.

RF_6**:** Form Template Method

Problem: You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.

Solution: Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.

RF_7**:** Inline Temp

Problem: You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.

Solution: Replace all references to that temp with the expression.

RF_8**:** Introduce Assertion

Problem: A section of code assumes something about the state of the program.

Solution: Make the assumption explicit with an assertion.

RF_9**:** Introduce Explaining Variable

Problem: You have a complicated expression.

Solution: Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

RF_{10}**:** Introduce Parameter Object

Problem: You have a group of parameters that naturally go together.

Solution: Replace them with an object.

RF_{11}**:** Parameterize Method

Problem: Several methods do similar things but with different values contained in the method body.

Solution: Create one method that uses a parameter for the different values.

RF_{12}**:** Preserve Whole Object

Problem: You are getting several values from an object and passing these values as parameters in a method call.

Solution: Send the whole object instead.

RF_{13}**:** Reduce Scope of Variable

Problem: You have a local variable declared in a scope that is larger than where it is used.

Solution: Reduce the scope of the variable so that it is only visible in the scope where it is used.

RF_{14}**:** Replace Array with Object

Problem: You have an array in which certain elements mean different things.

Solution: Replace the array with an object that has a field for each element.

RF_{15}**:** Replace Conditional with Polymorphism

Problem: You have a conditional that chooses different behavior depending on the type of an object.

Solution: Move each branch of the conditional to an overriding method in a subclass. Make the original method abstract.

RF_{16}**:** Replace Exception with Test

Problem: You are throwing an exception on a condition the caller could have checked first.

Solution: Change the caller to make the test first.

RF_{17}**:** Replace Method with Method Object

Problem: You have a long method that uses local variables in such a way that you cannot apply Extract Method.

Solution: Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

RF_{18}**:** Replace Nested Conditional with Guard Clauses

Problem: A method has conditional behaviour that does not make clear what the normal path of execution is.

Solution: Use Guard Clauses for all the special cases.

RF_{19}**:** Replace Parameter with Explicit Methods

Problem: You have a method that runs different code depending on the values of an enumerated parameter.

Solution: Create a separate method for each value of the parameter. **Note:** This transformation can be used to convert parameter adaptation to compositional adaptation

---

RF_{20}**:** Replace Temp with Query

Problem: You are using a temporary variable to hold the result of an expression.

Solution: Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.

RF_{21}**:** Separate Query from Modifier

Problem: You have a method that returns a value but also changes the state of an object.

Solution: Create two methods, one for the query and one for the modification.

RF_{22}**:** Split Temporary Variable

Problem: You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.

Solution: Make a separate temporary variable for each assignment.

RF_{23}**:** Substitute Algorithm

Problem: You want to replace an algorithm with one that is clearer.

Solution: Replace the body of the method with the new algorithm.

RF_{24}**:** Convert Local Variable to Field

Problem: You have local variable and you want to access it as a class member variable.

Solution: Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors.

Appendix B

Runtime Model Schema

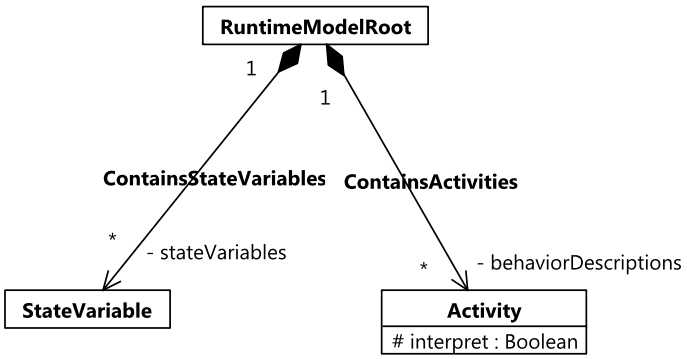

Figure B.1: The runtime model consists of state variables and activities

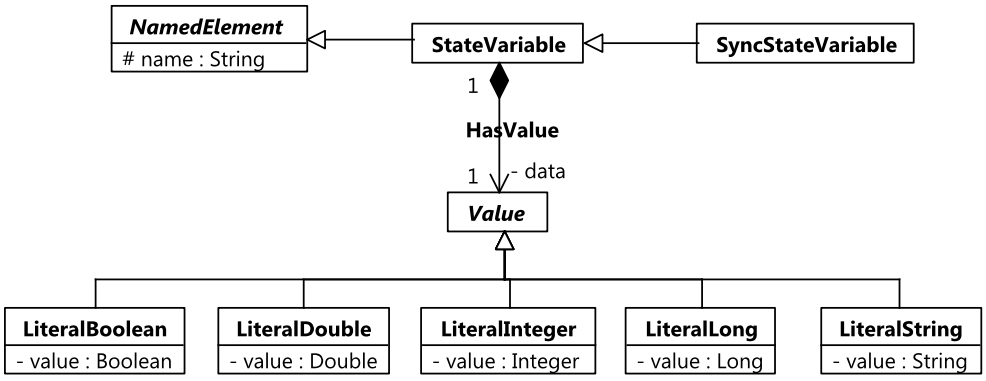

Figure B.2: The State Variables Used For Storing Sensed Data

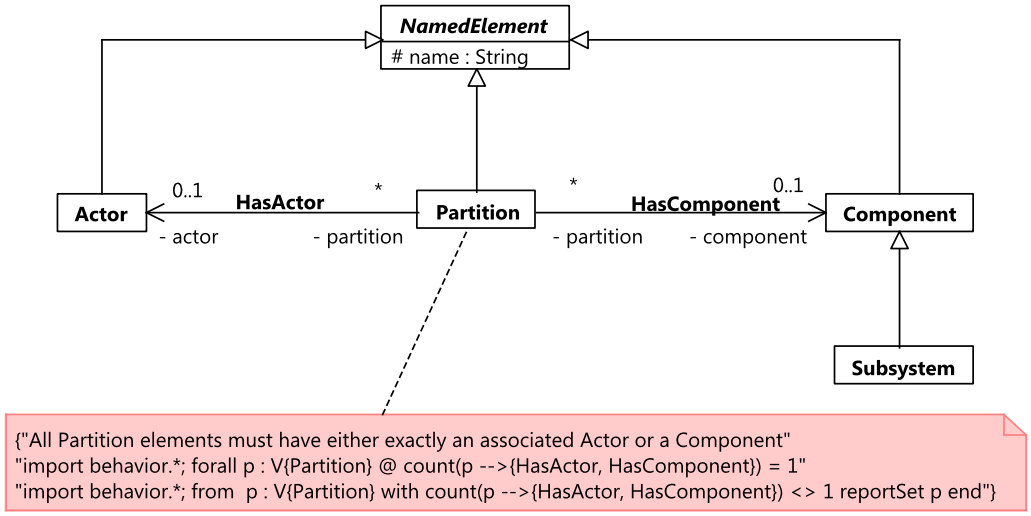

Figure B.3: A Detailed View On The Partition Element

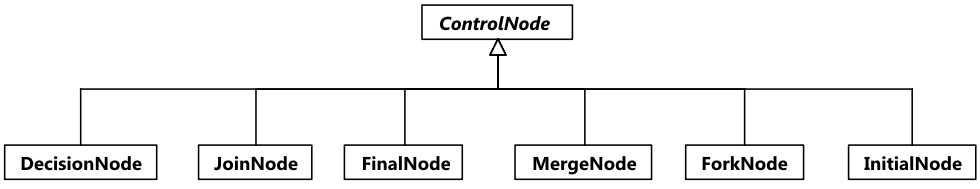

Figure B.4: The Essential Elements For Modeling Control Flow

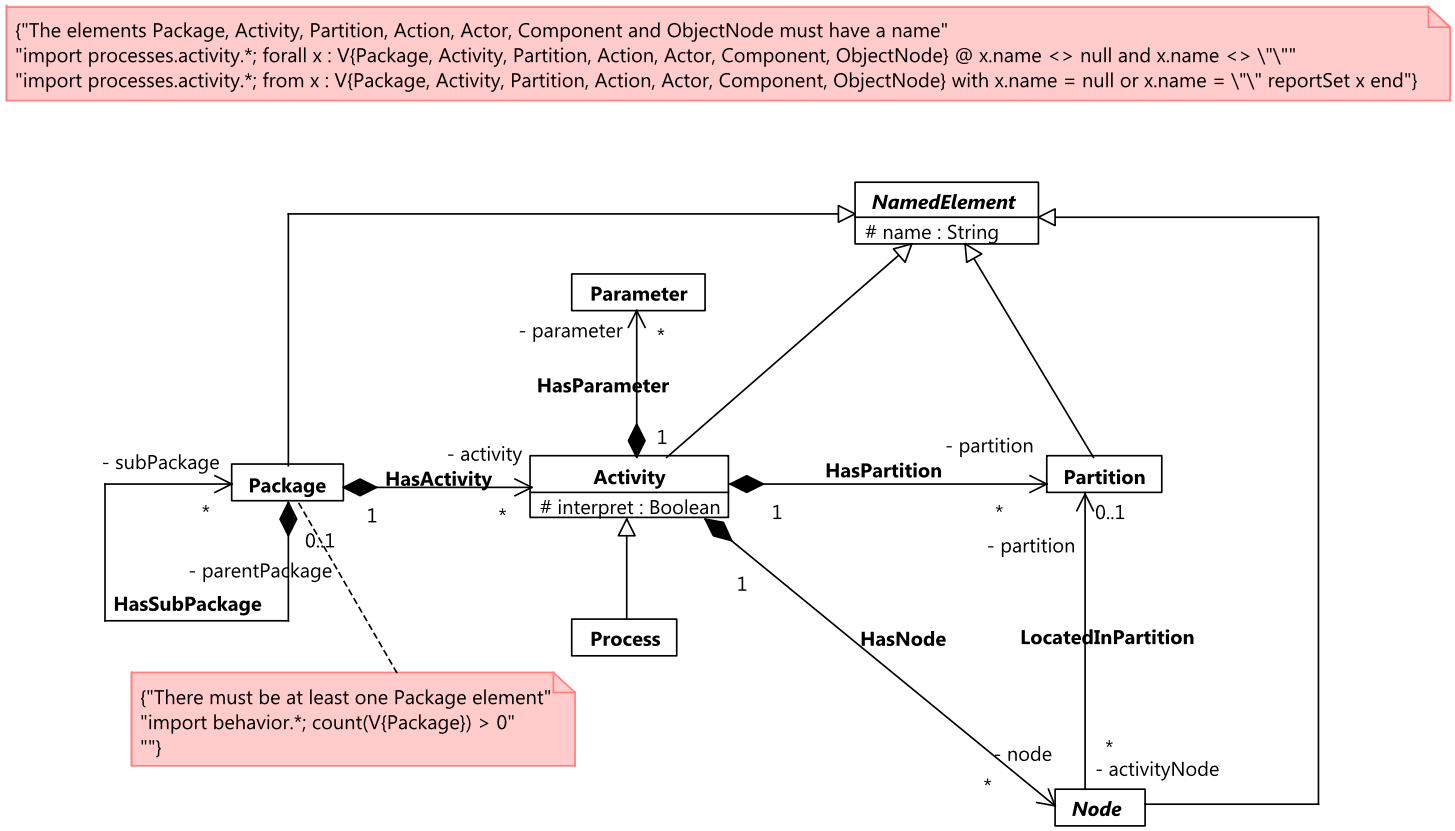

Figure B.5: The Activity With Related Elements

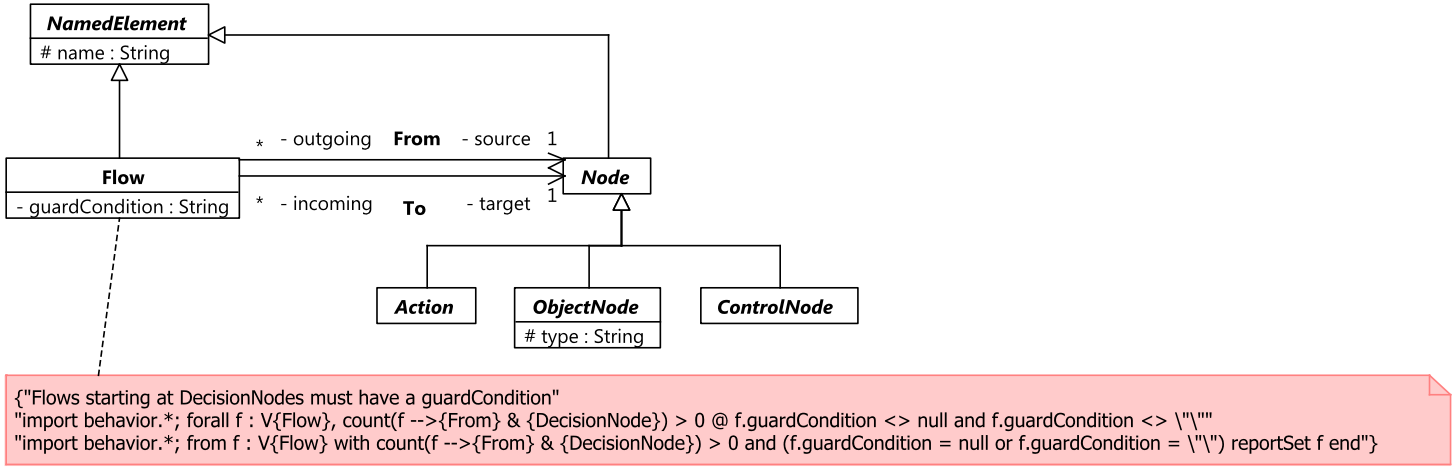

Figure B.6: Nodes Can Be Connected To Each Other Via A Flow

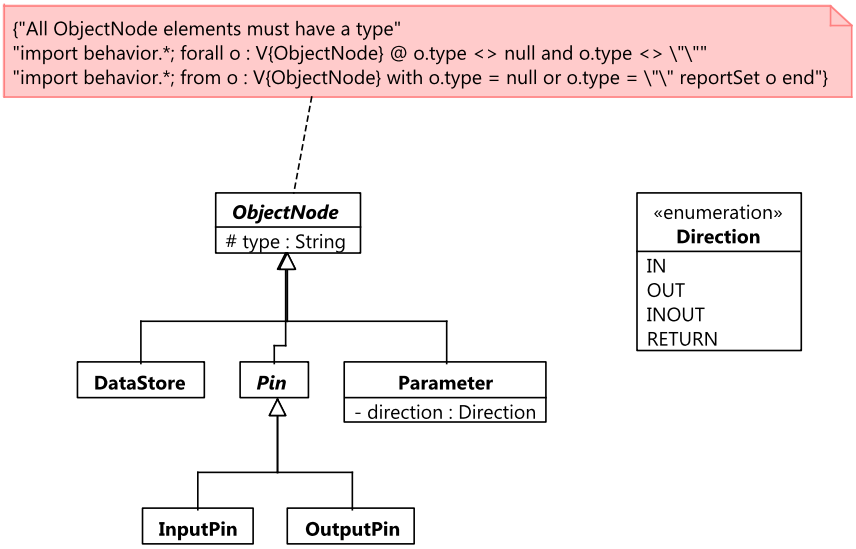

Figure B.7: Data Is Represented As Input And Output

References

[1] Robert Allen, Rmi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering*, pages 21–37, 1998. 34

[2] Mehdi Amoui, Mahdi Derakhshanmanesh, Jürgen Ebert, and Ladan Tahvildari. Software evolution towards model-centric runtime adaptivity. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 89–92, 2011. 9, 65

[3] Mehdi Amoui, Siavash Mirarab, Sepand Ansari, and Caro Lucas. A genetic algorithm approach to design evolution using design pattern transformations. *International Journal of Information Technology and Intelligent Computing*, 1(2):235–244, 2006. 23

[4] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive action selection in autonomic software using reinforcement learning. *Autonomic and Autonomous Systems, International Conference on*, 0:175–181, 2008. 10, 91, 127, 168

[5] Mehdi Amoui, Mazeiar Salehie, and Ladan Tahvildari. Temporal software change prediction using neural networks. *International Journal of Software Engineering and Knowledge Engineering*, 19(7):995–1014, 2009. 10, 111

[6] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture notes in Computer Science*, pages 27–47. 2009. 5

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on Software*, 28(10):970 – 983, 2002. 21

[8] ARM: Application Response Measurement. `http://www.opengroup.org/tech/management/arm/`, 2011. 27

[9] Reza Asadollahi, Maziar Salehie, and Ladan Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems*, pages 58–67, 2009. 28, 32, 82

[10] AspectJ: Crosscutting Objects for Better Modularity. `http://www.eclipse.org/aspectj/`, 2012. 28, 90

[11] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *Proceedings of 18th IEEE International Conference on Requirements Engineering*, pages 125–134, 2010. 53

[12] Robert Bell. Predicting the location and number of faults in large software systems. *IEEE Tranaction Software Engineering*, 31(4):340–355, 2005. 23

[13] Keith Bennett and Vaclav Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87, 2000. 3, 14

[14] Daniel Berry, Betty Cheng, and Ji Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering Foundation for Software Quality*, pages 95–100, 2005. 33

[15] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, 1999. 17

[16] Gordon Blair, Nelly Bencomo, and Robert France. Models@ run.time. *Computer*, 42(10):22–27, 2009. 92

[17] Barry W. Boehm. *Software engineering economics*, pages 641–686. Springer-Verlag New York, Inc., New York, NY, USA, 2002. 135, 160

[18] Shawn Bohner. Extending software change impact analysis into COTS components. In *Software Engineering Workshop.Proceedings of 27th Annual NASA Goddard/IEEE*, pages 175–182, 2002. 21

[19] Shawn Bohner and Robert Arnold. An Introduction To Software Change Impact Analysis. In *Software Change Impact Analysis*, pages 1–26. 1996. 20

[20] Philippe Boinot, Renaud Marlet, Jacques Noye, Gilles Muller, and Charles Consel. A declarative approach for designing and developing adaptive components. In *Proceedings of IEEE International Conference on Automated Software Engineering*, pages 111–119, 2000. 37

[21] Jeremy Bradbury, James Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings Of The 1St ACM SIGSOFT Workshop On Self-Managed Systems*, pages 28–33, 2004. 33

[22] Nevon Brake, James Cordy, Elizabeth Dan, Marin Litoiu, and Valentina Popes. Automating discovery of software tuning parameters. In *Proceedings of the International Workshop On Software Engineering for Adaptive And Self-Managing Systems*, pages 65–72, 2008. 28, 38

[23] Frederick Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987. 5, 67

[24] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986. 31

[25] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983. 20

[26] Greg Brown, Betty Cheng, Heather Goldsby, and Ji Zhang. Goal-oriented specification of adaptation requirements engineering in adaptive systems. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 23–29, 2006. 34

[27] Jens Bruhn, Christian Niklaus, Thomas Vogel, and Guido Wirtz. Comprehensive support for management of enterprise applications. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 755–762, 2008. 32

[28] Yuriy Brun, Giovanna Di, Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle amd Marin Litoiu, Mauro Pezz, and Mary Shaw. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009. 26

[29] BtM: IBM Build to Manage. `http://www.ibm.com/developerworks/eclipse/btm/`, 2011. 29

[30] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005. xvii, 3, 15, 16

[31] Jonathan Buckner, Joseph Buchta, Maksym Petrenko, and Vaclav Rajlich. JRipples: a tool for program comprehension during incremental change. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 149 – 152, may 2005. 139

[32] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley Software Patterns Series. John Wiley & Sons, Inc., 2007. 169

[33] Carlos Canal, Ernesto Pimentel, and José Troya. Specification and refinement of dynamic software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture*, pages 107–126, 1999. 34

[34] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and challenges in software reverse engineering. *Communications of the ACM*, 54(4):142–151, 2011. 19

[35] Walter Cazzola. Evaluation of object-oriented reflective models. In *In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems*, pages 386–387, 1998. 58

[36] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. *Software Evolution through Dynamic Adaptation of Its OO Design*, volume 2975 of *Lecture Notes in Computer Science*, pages 67–80. Springer Berlin/Heidelberg, 2004. 26

[37] Ned Chapin, Joanne Hale, Khaled Khan, Juan Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001. 3, 14

[38] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for*

Self-Adaptive Systems, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2009. 5

[39] Shang Cheng, An cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004. 25, 31, 69, 71

[40] Elliot Chikofsky and James Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 07(1):13–17, 1990. 18, 19

[41] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 2000. 35

[42] Lawrence Chung and Narry Subramanian. Process-oriented metrics for software architecture adaptability. In *Proceedings of 5th IEEE International Symposium Requirements Engineering*, pages 310–311, 2001. 36

[43] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998. 64

[44] James Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. 38

[45] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684 –702, 2009. 19

[46] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pages 1–17, 2003. 21, 22

[47] Elizabeth Dancy and James Cordy. STAC: Software Tuning Panels for Autonomic Control. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 146–160, 2006. 38, 62, 63

[48] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993. 34, 52

[49] Mahdi Derakhshanmanesh. Leveraging Model-Based Techniques for Runtime Adaptivity in Software Systems. Master's thesis, University of Koblenz-Landau, Germany, 2010. 85, 126

[50] Mahdi Derakhshanmanesh, Mehdi Amoui, Jürgen Ebert, and Ladan Tahvildari. GRAF: Graph-Based Runtime Adaptation Framework. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 128–137, 2011. xx, 68, 76

[51] Simon Dobson, Spyros Denazis, Antonio Fernndez, Dominique Gati, Erol Gelenbe, Fabio Massacci, Paddy Nixon Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, 2006. 34

[52] Theodosius Dobzhansky. What is an adaptive trait? *The American Naturalist*, 90(855):337–347, 1956. 1

[53] Theodosius G. Dobzhansky. *Genetics and the Origin of Species*. Columbia University Press, New York, 3rd edition, 1951. 1

[54] Larry Dooley. Case study research and theory building. *Advances in developing human resources*, 4(3):335–354, 2002. 133

[55] Jim Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, Department of Computer Science, Trinity College Dublin, 2004. 30, 31

[56] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of International Conference on Aspect-Oriented Software Development*, pages 66–73, 2004. 25, 37

[57] Jürgen Ebert and Daniel Bildhauer. Reverse Engineering Using Graph Queries. In *Proceedings of Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 335–362. 2010. 86

[58] Jurgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering. the tgraph approach. In *Proceedings of 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*, pages 67–81, 2008. 19

[59] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE in practice: A CASE for KOGGE. In *Proceedings of Advanced Information Systems Engineering*, volume 1250 of *Lecture Notes in Computer Science*, pages 203–216. 1997. 71

[60] Eclipse. Java Development User Guide. `http://www.eclipse.org/documentation/`, 2011. 112

[61] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. 99

[62] Eric Evans and Martin Fowler. Specifications. In *Proceedings of the Conference on Pattern Languages of Programming*, pages 97–34, 1997. 50

[63] Jean Marie Favre. Towards a basic theory to model model driven engineering. In *Workshop on Software Model Engineering*, 2004. 22

[64] Franck Fleurey and Arnor Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 606–621, 2009. 31

[65] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. 18, 22, 112, 159

[66] Fraps: Realtime Video Capture & Benchmarking. `http://www.fraps.com/`, 2011. 155

[67] Alen Ganek and Thomas Corbi. The dawning of the autonomic computing era. *IBM Systems Journal, Special Issues on Autonomic Computing*, 42(01):5–18, 2003. 35

[68] David Garlan, Shang Cheng, and Bradeley Schmerl. Increasing system dependability through architecture-based self-repair. In *Proceedings of Architecting Dependable Systems*, pages 61–89, 2003. 63

[69] David Garlan and Bradley Schmerl. Using architectural models at runtime: Research challenges. In *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 200–205. 2004. 41, 68, 92

[70] Hamoun Ghanbari and Marin Litoiu. Identifying implicitly declared self-tuning behavior through dynamic analysis. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 48 –57, may 2009. 139

[71] Tudor Gîrba and Stphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006. 23

[72] Heather Goldsby, Pete Sawyer, Nelly Bencomo, Betty Cheng, and Danny Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 36–45, 2008. 33

[73] Hassan Gomaa and Mohamed Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture*, pages 79 – 88, 2004. 169

[74] Paul Grace, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. A reflective framework for fine-grained adaptation of aspect-oriented compositions. In *Proceedings of the 7th international conference on Software composition*, pages 215–230, 2008. 25, 32, 37, 71

[75] Susan Graham, Peter Kessler, and Marshall Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982. 28

[76] Todd Graves, Alan Karr, and Harvey Siy Steve Marron. Predicting fault incidence using software change history. *IEEE Transactions Software Engineering*, 26(7):653–661, 2000. 23

[77] Philip Greenwood and Lynne Blair. Using dynamic aspect-oriented programming to implement an autonomic system. In *Proceedings of Dynamic Aspects Workshop*, pages 76–88, 2004. 25, 29

[78] Object Management Group. Unified Modeling Language (UML), Superstructure, 2010. Version 2.3. 78

[79] Carl Gunter, Elsa Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *Software, IEEE*, 17(3):37 –43, 2000. 42

[80] Charles Haley, Robin Laney, Jonathan Moffett, and Bashar Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(01):133–153, 2008. 53

[81] Robert Hanmer. *Patterns for Fault Tolerant Software*. John Wiley & Sons, Inc., 2007. 169

[82] Ahmed Hassan and Richard Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, 2005. 23

[83] Holger H. Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012. 3, 37

[84] Tassilo Horn and Jürgen Ebert. The gretl transformation language. In *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg, 2011. 22, 86

[85] Xishi Huang, Luiz Capretz, Jing Ren, and Danny Ho. A neuro-fuzzy model for software cost estimation. In *Proceedings of 3rd International Conference on Quality Software*, pages 126–133, 2003. 23

[86] IBM. Autonomic computing toolkit: Developer's guide. `www6.software.ibm.com/software/developer/library/autonomic/books/fpy3mst.htm`, 2005. 27

[87] IBM. An architectural blueprint for autonomic computing. `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`, 2006. 26, 80

[88] IEEE. Standard for software maintenance, 2006. 3, 14

[89] Michael Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley Publishing Co., 1995. 42, 45, 68

[90] JAIN SIP: The Standardized Java Interface to the Session Initiation Protocol. `https://jain-sip.dev.java.net/`, 2011. 138

[91] JAKE2. `http://bytonic.de/html/jake2.html`, 2012. xviii, 134, 147, 149

[92] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net. *International Journal on Software Tools for Technology Transfer*, 12(3):263–271, 2010. 22

[93] JBoss AOP: Framework for Organizing Cross Cutting Concerns. `http://www.jboss.org/jbossaop/`, 2011. 28, 89

[94] JBoss Application Server. `http://jboss.org/jbossas/`, 2011. 89

[95] Per Jönsson. *Impact Analysis Organisational Views and Support Techniques*. PhD thesis, Blekinge Institute of Technology, Sweden, 2005. 21

[96] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, 2006. 22

[97] Henry Kautz, David Mcallester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the 5th International Conference on the Principle of Knowledge Representation and Reasoning*, pages 374–384, 1996. 55

[98] Jeffrey Kephart and David Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. xix, 3, 4, 24, 29, 38, 66, 68, 84, 93, 164

[99] Jeffrey Kephart and William Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3–12, 2004. 54

[100] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. 111, 112

[101] Taghi Khoshgoftaar, Edward Allen, Kalai Kalaichelvan, and Nishith Goel. Predictive modeling of software quality for very large telecommunications systems. In *Communications of Conference Record Converging Technologies for Tomorrow's Applications*, volume 1, pages 214–219, 1996. 23

[102] Taghi Khoshgoftaar, Abhijit Pandya, and Hemant More. A neural network approach for predicting software development faults. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 83–89, 1992. 23

[103] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. 1997. 28

[104] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. 28

[105] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 21

[106] Mieczyslaw Kokar, Kenneth Baclawski, , and Yonet Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, 1999. 34, 43, 51

[107] Jeff Kramer and Jeff Magee. Analysing dynamic change in software architectures: a case study. In *Proceedings of 4th International Conference on Configurable Distributed Systems*, pages 91–100, 1998. 34

[108] Robert Laddaga. *Active Software*, volume 1936 of *Lecture Notes in Computer Science*, pages 11–26. Springer Berlin / Heidelberg, 2001. 24

[109] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994. 55

[110] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. In *Proceedings of Workshop on Design and Evolution of Autonomic Application Software*, pages 1–7, 2005. 36

[111] Gregory Lee, Martin Schulz, Dong Ahn, Andrew Bernat, Bronis de Supinski, Steven Ko, and Barry Rountree. Dynamic binary instrumentation and data aggregation on large scale systems. *International Journal of Parallel Programming*, 35(3):207–232, 2007. 28

[112] Meir Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, 1996. 2, 4, 14, 93, 164

[113] Meir Lehman and Juan Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002. 14

[114] Karl Lieberherr and Cun Xiao. Customizing adaptive software to object-oriented software using grammars. *International Journal of Foundations of Computer Science*, 5(5):179–208, 1994. 25

[115] Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. 14, 15

[116] Sam Lightstone. Foundations of autonomic computing development. In *Proceedings of 7th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, pages 163–171, 2007. 35

[117] David Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In *Proceedings of 1st workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, 1986. 20

[118] Michael Littman, Nishkam Ravi, Eitan Fenson, and Rich Howard. Reinforcement learning for autonomic network repair. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 284–285, 2004. 31

[119] Xia Liu and Qing Wang. Study on application of a quantitative evaluation approach for software architecture adaptability. In *Proceedings of the 5th International Conference on Quality Software*, pages 265–272, 2005. 37

[120] Jochen Ludewig. Models in software engineering - an introduction. *Software and Systems Modeling*, 2:5–14, 2003. 41, 67

[121] Michael R. Lyu, editor. *Handbook of software reliability engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996. 138

[122] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Notes*, 22(12):147–155, 1987. 24, 28

[123] Pattie Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6(1-2):49–70, 1990. 30

[124] Thomas McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308 – 320, 1976. 135

[125] Philip McKinley, Masoud Sadjadi, Eric Kasten, and Betty Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004. 37, 58, 108

[126] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152(0):125 – 142, 2006. 21

[127] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. 111, 112

[128] Michael Merideth and Priya Narasimhan. Retrofitting networked applications to add autonomic reconfiguration. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005. 38

[129] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997. 123

[130] Audris Mockus, Basking Zhang, and Paul Luo Li. Predictors of customer perceived software quality. In *Proceedings of the 27th international conference on Software engineering*, pages 225–233, 2005. 23

[131] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009. 31, 41, 68, 92

[132] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132, 2009. 32, 69

[133] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of control in adaptive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, pages 23–26, 2008. 57, 63

[134] Emerson Murphy-Hill. *Programmer Friendly Refactoring Tools*. PhD thesis, Portland State University, 2009. 112

[135] John Mylopoulos, Lawrence Chung, and Eric Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, 1999. 52

[136] George Stephanides Nikolaos Tsantalis, Alexander Chatzigeorgiou. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, 2005. 23

[137] Object Management Group. MDA Guide Version 1.0.1, 2003. 22

[138] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, 2011. 22

[139] Object Management Group. MOF 2 XMI Mapping (XMI), Version 2.4.1, 2011. 86

[140] Object Management Group. UML profile specifications. `www.omg.org/technology/documents/profile_catalog.htm`, 2011. 66

[141] Katsuhiko Ogata. *Modern control engineering*. Prentice Hall PTR, 2001. xix, 26, 34, 43, 101

[142] William Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. 22, 111

[143] OpenJSIP: Open-Source SIP Services Implemented in Java. `http://code.google.com/p/openjsip/`, 2011. 134, 137

[144] Peyman Oreizy, Michael Gorlick, Richard Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David Rosenblum, and Alexander Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999. 24, 37, 38

[145] Peyman Oreizy, Nenad Medvidovic, and Richard Taylor. Architecture-based runtime software evolution. In *Proceedings of International Conference on Software Engineering*, pages 177–186, 1998. 25, 137

[146] Michael Oudshoorn, Muztaba Fuad, and Debzani Deb. Towards autonomic computing: Injecting self-organizing and self-healing properties into java programs. In *Proceedings of the 5th conference on New Trends in Software Methodologies Tools and Techniques*, volume 147, pages 384–406, 2006. 38

[147] OW2 Consortium. ASM: Java bytecode manipulation and analysis framework. `http://asm.ow2.org/`, 2010. 28

[148] Janak Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006. 38

[149] David Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995. xix, 46, 47

[150] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC: an aspect-based distributed dynamic framework. *Software: Practice and Experience*, 34(12):1119–1148, 2004. 29

[151] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, 2002. 28

[152] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA, 2002. 147

[153] Sheela Ramanna. Rough neural network for software change prediction. In *Proceedings of International Conference on Rough Sets and Current Trends in Computing*, pages 602–609, 2002. 23

[154] Andres Ramirez and Betty Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 49–58, 2010. 169

[155] Rational Software Architect. `http://www.ibm.com/developerworks/rational/products/rsa/`, 2010. 86

[156] Dave Robson, Keith Bennett, Barry Cornelius, and Malcolm Munro. Approaches to program comprehension. *Journal of Systems and Software*, 14(2):79 – 84, 1991. 19

[157] Julio Rosenblatt. DAMN: a distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):339–360, 1997. 30

[158] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. 55

[159] Seyed Masoud Sadjadi, Philip McKinley, Betty Cheng, and Kurt Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 1243–1261, 2004. 32

[160] Mazeiar Salehie. *A Quality-Driven Approach to Enable Decision-Making in Self-Adaptive Software*. PhD thesis, University of Waterloo, 2009. 100, 137

[161] Mazeiar Salehie, Sen Li, Reza Asadollahi, and Ladan Tahvildari. Change support in adaptive software: A case study for fine-grained adaptation. In *Proceedings of the 6th IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 35–44, 2009. 29

[162] Mazeiar Salehie and Ladan Tahvildari. Autonomic computing: emerging trends and open problems. In *Proceedings of the ICSE Workshop on Design and Evolution of Autonomic Application Software*, pages 82–88, 2005. xix, 36, 51

[163] Mazeiar Salehie and Ladan Tahvildari. Coordinating self-healing and self-optimizing in autonomic elements: an experiment. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 98, 2006. 37

[164] Mazeiar Salehie and Ladan Tahvildari. A weighted voting mechanism for action selection problem in self-adaptive software. In *Proceedings of 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 328–331, 2007. 31

[165] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009. 31, 34, 35, 63, 82

[166] Mohammed Salifu, Yijun Yu, and Bashar Nuseibeh. Specifying monitoring and switching problems in context. In *Proceedings of the 15th IEEE International Requirements Engineering Conference*, pages 211–220, 2007. 33

[167] Robert Seacord, Daniel Plakosh, and Grace Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 3, 15, 17

[168] Mary Shaw. Prospects for an engineering discipline of software. *Software*, 7(6):15–24, 1990. 2

[169] Martin Shepperd and Gada Kadoda. Comparing software prediction techniques using simulation. *IEEE Transactions on Software Engineering - Special section on the seventh international software metrics symposium*, 27(11):1014–1022, 2001. 23

[170] Shigeru Chiba. Javassist: Java bytecode manipulation made simple, 2010. 28

[171] SIPp: Traffic Generator for the SIP Protocol. `http://sipp.sourceforge.net/`, 2010. 143

[172] Wassiou Sitou and Bernd Spanfelner. Towards requirements engineering for context adaptive systems. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, volume 2, pages 593–600, 2007. 33

[173] Vitor Souza, Alexei Lapouchnian, William Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pages 60–69, 2011. 53

[174] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 196–205, 1994. 28

[175] Nary Subramanian and Lawrence Chung. Metrics for software adaptability. In *Proceedings of the Software Quality Management Conference*, pages 371–380, 2001. 37

[176] Nary Subramanian and Lawrence Chung. Software architecture adaptability: An NFR approach. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 52–61, 2001. 36, 46

[177] Nary Subramanian and Lawrence Chung. Tool support for engineering adaptability into software architecture. In *Proceedings of the 5th International Workshop on Principles of Software Evolution*, pages 86–96, 2002. 36

[178] Sun Microsystems, Inc. *Java Management Extensions (JMX) Specification, version 1.4*, 2006. 28

[179] Sun Microsystems, Inc. *Java Virtual Machine Tool Interface (JVMTI)*, 2006. 28

[180] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. 31, 55, 168

[181] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, 2000. 33

[182] Ladan Tahvildari and Kostas Kontogiannis. Improving design quality using meta-pattern transformations: A metric-based approach. *Software Maintenance and Evolution: Research and Practice*, 16(4-5):331–361, 2004. 22

[183] Pentti Tarvainen. Adaptability evaluation of software architectures; a case study. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, volume 2, pages 579–586, 2007. 37

[184] BCEL Team. BCEL: Byte Code Engineering Library. `http://jakarta.apache.org/bcel/`, 2006. 28

[185] Gerald Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30, 2007. 31, 55

[186] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1):89–120, 2001. 22, 111

[187] TPTP: Eclipse Test & Performance Tools Platform. `http://www.eclipse.org/tptp/`, 2011. 155

[188] Wladyslaw Turski. Software stability. In *Proceedings of the 6th ACM Europian Regional Conference on System Architecture*, pages 107–116, 1981. 14

[189] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley and Sons, 2009. 52, 54, 100

[190] Daniel Varr and Andres Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214 – 234, 2007. 22

[191] Norha Villegas, Hausi Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pages 80–89, 2011. 51

[192] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57(0):109 – 143, 2001. 21

[193] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 39–48, 2010. 32, 71, 92

[194] Anneliese von Mayrhauser and Marie Vans. Industrial experience with an integrated code comprehension model. *Software Engineering Journal*, 10(5):171 –182, 1995. 19

[195] Anneliese Von Mayrhauser and Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44 –55, 1995. 19

[196] Nelson Weiderman, John Bergey, Dennis Smith, and Scott Tilley. Approaches to legacy system evolution. Technical Report CMU/SEI-97-TR-014, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, 1997. 14, 17

[197] Peter Weissgerber and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions in Software Engineering*, 31(6):429–445, 2005. 23

[198] Danny Weyns, Sam Malek, and Jesper Andersson. Forms: a formal reference model for self-adaptation. In *Proceedings of the 7th international conference on Autonomic computing*, pages 205–214, 2010. 24

[199] Ji Zhang and Betty Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, pages 371–380, 2006. 34

[200] Ji Zhang and Betty Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361–1369, 2006. 34