Domain-specific modeling language for self-adaptive software system architectures

Filip Křikava

L'UNIVERSITÉ NICE-SOPHIA ANTIPOLIS – UFR SCIENCES
École Doctorale de Sciences et Technologies
de l'Information et de la Communication

T H È S E

pour obtenir le titre de
Docteur en Sciences
de l'Université Nice-Sophia Antipolis

Discipline : Informatique

présentée et soutenue par
Filip Křikava
le 22 Novembre 2013

Domain-Specific Modeling Language for Self-Adaptive Software System Architectures

Thèse dirigée par Philippe Collet et Johan Montagnat et préparée au sein du laboratoire I3S, équipe MODALIS

Jury

Philippe Collet	Professeur, Université Nice-Sophia Antipolis	Co-Directeur
Johan Montagnat	Directeur de Recherche, I3S - CNRS	Co-Directeur
Bernhard Rumpe	Professeur, RWTH Aachen University	Rapporteur
Lionel Seinturier	Professeur, Université Lille 1	Rapporteur
Jacques Malenfant	Professeur, Université Pierre et Marie Curie	Examinateur
Michel Riveill	Professeur, Université Nice Sophia Antipolis	Examinateur

Abstract

The vision of Autonomic Computing and Self-Adaptive Software Systems aims at realizing software that autonomously manage itself in presence of varying environmental conditions. Feedback Control Loops (FCL) provide generic mechanisms for self-adaptation, however, incorporating them into software systems raises many challenges.

The first part of this thesis addresses the integration challenge, *i.e.*, forming the architecture connection between the underlying adaptable software and the adaptation engine. We propose a domain-specific modeling language, FCDL, for integrating adaptation mechanisms into software systems through external FCLs. It raises the level of abstraction, making FCLs amenable to automated analysis and implementation code synthesis. The language supports composition, distribution and reflection thereby enabling coordination and composition of multiple distributed FCLs. Its use is facilitated by a modeling environment, Actress, that provides support for modeling, verification and complete code generation. The suitability of our approach is illustrated on three real-world adaptation scenarios.

The second part of this thesis focuses on model manipulation as the underlying facility for implementing Actress. We propose an internal Domain-Specific Language (DSL) approach whereby Scala is used to implement a family of DSLs, Sigma, for model consistency checking and model transformations. The DSLs have similar expressiveness and features to existing approaches, while leveraging Scala versatility, performance and tool support.

To conclude this thesis we discuss further work and further research directions for MDE applications to self-adaptive software systems.

Résumé

Le calcul autonome vise à concevoir des logiciels qui prennent en compte les variations dans leur environnement d'exécution. Les boucles de rétro-action (FCL) fournissent un mécanisme d'auto-adaptation générique, mais leur intégration dans des systèmes logiciels soulève de nombreux défis.

Cette thèse s'attaque au défi d'intégration, c.à.d. la composition de l'architecture de connexion reliant le système logiciel adaptable au moteur d'adaptation. Nous proposons pour cela le langage de modélisation spécifique au domaine FCDL. Il élève le niveau d'abstraction des FCLs, permettant l'analyse automatique et la synthèse du code. Ce langage est capable de composition, de distribution et de réflexivité, permettant la coordination de plusieurs boucles de rétro-action distribuées et utilisant des mécanismes de contrôle variés. Son utilisation est facilitée par l'environnement de modélisation Actress qui permet la modélisation, la vérification et la génération du code. La pertinence de notre approche est illustrée à travers trois scénarios d'adaptation réels construits de bout en bout.

Nous considérons ensuite la manipulation de modèles comme moyen d'implanter Actress. Nous proposons un Langage Spécifique au Domaine interne qui utilise Scala pour implanter une famille de DSLs. Il permet la vérification de cohérence et les transformations de modèles. Les DSLs résultant ont des propriétés similaires aux approches existantes, mais bénéficient en plus de la souplesse, de la performance et de l'outillage associés à Scala.

Nous concluons avec des pistes de recherche découlant de l'application de l'IDM au domaine du calcul autonome.

I almost wish I hadn't gone down that rabbit hole - and yet - and yet - it's rather curious, you know, this sort of life!

--- Alice

Contents

List of Figures

Listings

CHAPTER 1

Introduction

1.1 Context and Motivation

The past two decades have witnessed the proliferation of computing devices alongside with the development of their raw computing capacities growing at exponential rates. While only a few years ago, computer systems had occupied a well delimited part of our lives, today, this is not true anymore as they are present everywhere [Maggio, 2011]. This phenomenal growth together with the expansion of the Internet have opened a new era of information accessibility [IBM, 2006]. However, this boom has been also accompanied with a steep grow of complexity of computing systems, putting enormous demand on the underlying information technology infrastructure that is now reaching the edge of being manageable. In his keynote to National Academy of Engineers at Harvard University in October 2001, IBM senior research vice-president Paul Horn, said: ``*In fact, the growing complexity of the IT infrastructure threatens to undermine the very benefits information technology aims to provide*'' [Horn, 2001].

To give some examples of this complexity increase, consider the following two aspects of complexity increase: the expansion of the global IP traffic in data centers as a consequence of growing IT infrastructures and the size growth of the software source code. From 2011 the global data center IP traffic has been growing at annual rate of 31% nearly quadrupling from 1.7 ZB to 6.6 ZB by the end of 2016 [Cisco Systems, 2012]. The code size of the most popular HTTP server [Netcraft, 2013], Apache, has been growing steadily about 100K lines-of-code per year over last 7 years; in the case of Linux Kernel it is more than 1M [Ohloh, 2013] lines a year.

The complexity increase is not only in the infrastructure and the development, but also more and more in the operation management. The cost associated with keeping IT systems in production are growing steadily. These IT systems have become labor intensive and the administrative expenses made up almost entirely of people costs represents from 60% to 80% of the total cost of information system ownership [CRA, 2003, Sherry et al., 2012]. Moreover, complex software systems are difficult to configure and manage.

For example, the number of configuration directives in Apache HTTP server has almost doubled from 178 in the version 1.3 to 341 in the 2.4 version. The cost associated with troubleshooting mis-configured systems is substantial accounting for about 17% of the total cost of ownership [Attariyan and Flinn, 2010].

So far, the complexity has been managed by skilled IT professionals, but we can already see that this approach does not scale. Not only because of the increasing costs of development and operations, but most importantly, it is the demand for the skilled professional that is already outstripping its supply [IBM, 2006]. Moreover, the administration challenges are reaching the point of exceeding the capabilities of human operators [Müller et al., 2009]. This is becoming a significant limiting factor for further growth of information systems and points to an inevitable need for new operation modes to be implemented. Such operation modes that would make systems smarter, resilient to dynamic changes and automate their functions in accordance with higher-level policies [Feiler et al., 2006].

Towards Self-Adaptive Software Systems In the last decade, a lot of effort has been invested into addressing this rapid complexity growth by using technology to manage technology. The vision is to allow systems to self-manage themselves in order to reduce the total cost of ownership of complex IT systems [Ganek and Corbi, 2003]. This approach sought as *Autonomic Computing* [Kephart and Chess, 2003] or *Self-Adaptive Software Systems* [Cheng et al., 2009a] aims at realizing computing systems and applications that manage themselves autonomously, with minimal or no human interactions. Such systems then provide some self-management properties, mainly *self-configuration*, *self-healing*, *self-optimization* and *self-protection*. Autonomic[1] systems are thus characterized by their ability to detect, devise and apply adaptations when needed in order to follow some higher-level goals.

A common approach to engineer these systems is to use *Feedback Control Loops* (FCL) that provide generic mechanisms for self-adaptation [Brun et al., 2009]. In its most general form a FCL is composed of three phases: monitoring, decision-making, and reconfiguration. Monitoring observes the state of the target system and its environment; decision-making uses observations to determine what actions, among the set of all permissible actions, should be taken in order to get the system into a desired state; and reconfiguration performs the actual changes to the application structure or behavior. While the principles behind a FCL are rather simple, the development of a self-adaptive system is not. There are many different challenges that have to be overcome in their engineering [Salehie and Tahvildari, 2009, Brun et al., 2009, Cheng et al., 2009a]. Among them, we are interested in the issues related to designing self-adaptive systems which can be decomposed into:

1. designing the underlying adaptable software system (the *target system*),
2. designing the adaptation engine, and
3. forming the architecture connecting them together.

[1] The origin of autonomic comes from the Greek autonomous, *αυτ νομος*, meaning *self-governing*.

The first issue is related to the design for adaptability since for a software to become adaptable it needs to provide (or allow it to be instrumented to provide) necessary touch-points, *i.e.*, interfaces exposing running system state and management operations. This requires significant knowledge about the running system and its API and it might be a particularly challenging problem for the legacy systems.

Assuming, the target system provides the necessary means to be observed and modified, the second issue to be solved is related to finding the appropriate control model to drive the adaptation itself. In its most basic view, decision making merely consists in observing the state of a system and choosing an action, among the set of admissible actions for the state. However, it is far from being trivial as it requires a significant amount of domain-specific knowledge and involves an extensive experimentation with a variety of FCL procedures and tuning parameters [Hellerstein et al., 2004]. The use of control theory is often envisaged as a viable solution to design self-adaptive computing systems [Maggio, 2011]. However, there are still unsolved issues that prevents control-theoretical design to be widely adopted (*e.g.* a lack of systematic approaches to testing controller implementations) [Hellerstein, 2010]. Moreover, in the computer science context alone there are no well-assessed techniques to design such control loops [Maggio, 2011].

Finally, assuming an adaptable target system with designed controllers, the last issue concerns the interoperability and system-wide architecture of the complete self-adaptive software system. Implementing and integrating the control model into the underlying target system is an error-prone and a difficult task. In non-trivial cases, the integration task includes ensuring that monitoring data are consistently collected across all the sources, and that adaptation actions are coordinated. Furthermore, while the control model is usually prototyped in a domain-specific modeling environment such as MATLAB[2] [Hellerstein et al., 2004], often, in the end, the adaptation engine has to be integrated into the final executable system using some *General Purpose Programming Language* (GPL) such as C, C++ or Java. Typically, this integration is done manually by researchers or engineers requiring an extensive handcrafting of a non-trivial implementation code, particularly in the case of remotely distributed systems. Moreover, the integration code has to deal with other software-development related aspects such as error handling and related non-functional properties. This is both a cumbersome and an error-prone activity. As a result, these handcrafting of implementations gives rise to significant *accidental complexities*[3], *i.e.*, complexities that are caused by the approach chosen to solve the problem [Brooks Jr., 1987] and which are already considerable with the complexity of contemporary software systems.

Towards Model-Driven Software Development One of the major problem associated with the development of self-adaptive systems is the wide conceptual gap between the problems being addressed and their actual software implementations. A promising ap-

[2] http://www.mathworks.com/products/matlab

[3] Sometimes, a term *incidental complexity* is used instead in order to stress the fact that the complexity is ``*happening in connection with or resulting from something more important*'' rather than ``*occurring by chance, unexpectedly, or unintentionally*'', Collins English Dictionary, HarperCollins Publishers, 2003.

proach to bridge this gap lies in *Model-Driven Engineering* (MDE) techniques. The vision behind MDE is in using models as the primary development artifacts to describe complex systems at multiple levels of abstraction and perspectives effectively expressing the domain concepts [France and Rumpe, 2007]. In particular, *Domain-Specific Modeling* (DSM) can significantly raise the level of abstraction allowing one to specify solutions directly using problem-level abstractions [Kelly and Tolvanen, 2008]. MDE relies on systematic modeling and model transformations that analyze certain aspects of models and then synthesize various types of artifacts, such as source code [Schmidt, 2006]. By automating synthesis of running systems the accidental complexities can be reduced consequently allowing developers, in case of self-adaptive software systems, to focus rather on developing and experimenting with alternative FCL designs.

An essential part of MDE are model manipulation languages and tools. They provide support for automating the MDE operations such as model consistency checking and transformation. While, and still it is often the case, the GPLs can be used for implementing the model manipulation tasks, the resulting code is less readable and verbose, with the expressed concern being lost among other commands again gives rise to *accidental complexities*. Indeed, the inability of some GPLs to alleviate this complexity by expressing domain concepts effectively [Schmidt, 2006] is one of the reasons why a growing number of different model manipulation *Domain-Specific Languages* (DSL) have been proposed in the past years. These DSLs allow developers to manipulate models using higher-level abstractions resulting in gains in expressiveness and ease of use when compared with the use of GPLs [Mernik et al., 2005]. On the other hand, evolving DSLs to a sufficient degree of maturity and providing them with modern tools including IDEs, debuggers, profilers require significant effort [Chafi et al., 2010]. Furthermore, a model manipulation DSL usually do not exist in isolation. An MDE typically involves a chain of model management tasks and the presence of multiple model manipulation languages therefore raises the issues of integration, interoperability, consistency and reuse [Kolovos et al., 2008b]. These issues together with shortcomings that concerns provided tool support (*e.g.* performance and stability) contribute to the *accidental complexities*, albeit of a different nature to those associated with the use of GPLs.

1.2 Objectives

With respect to the discussed situation, the context of the research thesis is as follows:

Despite the advances in self-adaptive software systems engineering, there is still a need for an integrated approach that would reduce the accidental complexities associated with forming the architecture connecting an adaptation engine into an adaptable target system. Therefore, the main objective of this thesis is *to present an approach that facilitates developers in integrating self-adaptive mechanisms into software systems through external Feedback Control Loops*. To address the accidental complexities outlined above in this thesis we define two main goals:

1. *Provide researchers and engineers with a flexible abstraction and extensible tool support fa-*

cilitating its use to allow them to experiment and to put easily self-adaptation in practice, without the need for coping with low-level implementation and infrastructure details.

2. *Develop lightweight model manipulation languages as internal domain-specific languages that would have similar expressiveness and features as the ones found in existing model manipulation languages, while providing good performance and usable tool support.*

We then decompose the first goal into the following objectives:

1.1. Create a flexible model for feedback control architectures that allows fine grain decomposition of loop elements and raises the level of abstraction at which these elements are described.

1.2. Provide a set of model-transformation tools to facilitate the use of the model and to automate the development process where possible.

Similarly, the second goal is split into these objectives:

2.1. Provide a common integration layer between an existing meta-modeling framework and a host general-purpose programming language for the essential modeling operation such as navigation, modification and delegation.

2.2. Using this layer, create task-specific model manipulation languages for model consistency checking, model-to-model and model-to-text transformation.

1.3 Contributions

The two goals set for this thesis consequently leads to two main contributions.

First, it is the model-driven software development approach providing a systematic way to integrate self-adaptation capabilities into software systems based on external feedback control loops. Concretely, we propose a domain-specific modeling language offering flexible high level abstraction for describing FCL architectures as a hierarchically composed networks. The elements in these networks represent both the individual processes of the FCLs as well as the target systems via their introspection and management interfaces. The language supports distribution and reflection thereby enabling coordination of multiple remotely distributed control loops using various control schemes. The use of the language is facilitated by a modeling environment that allows for a modular specification and configuration of both structural and behavioral parts of the system aiming at complex, reusable and coordinated control loops. The environment further supports verification of architecture consistency including user-defined invariants as well as connectivity and data reachability properties through external verification. The resulting architecture models are used as inputs into a source code generator that synthesizes complete executable software artifacts so the described control loop can be instantiated and run.

The second contribution of this thesis is related to the model manipulation techniques. We propose a family of internal domain-specific languages in Scala [Odersky et al., 2004]

programming language for EMF (*Eclipse Modeling Framework*) [Steinberg et al., 2008] model manipulations. Built on a common model navigation and modification support, we provide a set of task-specific languages for model consistency checking and model-to-model and model-to-text transformations. The seamless integration of the host language with the modeling framework allows realizing DSLs that are consistent and have similar expressiveness and features that are found in existing model manipulation languages while leveraging from Scala versatility, performance and tool support.

1.4 Outline

The document is composed of nine chapters, organized into three parts. The first part concerns self-adaptive software systems and reports on the proposed solution for integrating self-adaptive mechanisms into software systems. The second part addresses the challenges regarding model manipulation languages . Finally, the third part provides an evaluation of the two preceding parts summarizing their contributions and outlining further work and directions for further research.

Chapter 2 presents the state of the art of self-adaptive software systems. It introduces their background, reviews the principles behind feedback control loops and its application to software. Then it gives an overview of related work and provides motivation of our objectives. Parts of this chapter were published in [Collet et al., 2010].

Chapter 3 is the first chapter of our contribution. It gives the rationale behind the selected approach and introduces a running adaptation scenario that is used throughout the thesis to better illustrate the approach. The main part of this chapter details the abstract syntax of our domain-specific modeling language and presents its graphical notation. This chapter is based on the work published in [Krikava and Collet, 2011, Krikava et al., 2011, 2012].

Chapter 4 complements the abstract syntax of the domain-specific language with a model of computation and a notion of interaction contracts that together define operational rules governing interactions among the feedback processes. This chapter was partially published in [Krikava and Collet, 2011, Krikava et al., 2011, 2012].

Chapter 5 presents the implementation of the language and the associated tool support for modeling, code generation and verification including their integration into a development environment. This chapter was partially included in[Krikava et al., 2012, Krikava and Collet, 2012b].

Chapter 6 starts the second part of this thesis that focuses on model manipulation. It presents the state-of-the-art model manipulation languages and tools, discusses their limitations and motivates our internal DSL approach to model manipulation. This chapter shares content with [Krikava and Collet, 2012a].

Chapter 7 presents our approach to model manipulation implemented as a family of internal DSLs for model consistency checking and model transformations. Each language is demonstrated through a series of examples. This chapter was partially published in [Krikava and Collet, 2012a].

Chapter 8 provides an evaluation of the two approaches. Two additional end-to-end case studies are implemented to further demonstrate the capability of the domain-specific modeling language and the modeling environment. This is followed by a discussion about applications, capabilities and an overview of the software quality attributes of both contributions. Some of the results of this chapter were part of [Krikava and Collet, 2011, 2012a].

Chapter 9 summarizes the dissertation and recalls its contributions. It outlines future work and concludes with a discussions on longer-term perspectives and further research directions.

Throughout this thesis we use the Scala programming language [Odersky et al., 2004] to illustrate the described concepts.

Part I

On Engineering Self-Adaptive Software Systems

CHAPTER 2

Self-Adaptive Software Systems

Self-adaptation is a general mechanism of a behavioral adjustment in a response to an environmental change and it is applicable across a wide range of systems. In this thesis we focus solely on its application in the software domain[1] referred to as *self-adaptive software systems* [Cheng et al., 2009a].

In this chapter we describe the context and give an overview of the state of the art of the self-adaptive software system engineering related to the approach followed by this thesis. That is, integration of self-adaptive mechanisms into software systems through external feedback control loops. We start by a general introduction to the principles of self-adaptation in software systems, present feedback control loop and its application to software followed by the concepts of autonomic computing. Next, we outline some of the existing approaches for engineering these systems. At the end we discuss the issue related to the integration of self-adaptive mechanisms into software systems.

2.1 Principles

Traditionally, software systems have been designed as open-loop systems primarily focusing on the correctness of their functions. Indeed, over the years, the software engineering discipline has been concerned with tools and methods facilitating the development of functionally correct software [Abdelzaher et al., 2003], emphasizing the static structure of the system and, to some extent, neglecting their dynamics aspects [Brun et al., 2009]. Once a system is designed and deployed to deliver certain functionality to its users, the functional properties remain mostly unchanged. Therefore, a correct software system operation usually requires human supervision. In cases of its malfunction, *i.e.*, when the system goes beyond its acceptable operating bounds, the responsibility of troubleshooting and fixing problems relies on a human administrator. Besides that human administrators are prone to long reaction time largely depending on their expertise, the course of

[1] Self-adaptation might be present at different layers along system domain dimensions for example it can be also embedded in the hardware layer (*e.g.*, self-diagnosing or self-tuning devices)

actions often implies restarting a part or the entire system leading to an unwanted system downtime [Garlan et al., 2004].

Because of the complexity of the contemporary software systems, especially the systems dealing with distributed applications and environment uncertainty, and costs associated with operations and system outages, there is a high-demand for management automation. This demand has led scientists and engineers to look for alternative ways to design and manage software, seeking an inspiration in diverse related fields such as control theory, biology, robotics and artificial intelligence [Cheng et al., 2009a]. One of the most promising research directions in this endeavor is in *self-adaptation* where systems are enriched with the capacity to autonomously adjust their behavior in response to observed changes in their contexts. Traditionally, self-adaptive systems have been used in a number of different areas. While it is not new in the software domain, it has been only recently that supporting some form of self-adaptation has become a critical requirement of software systems.

Similarly to the advent of the Internet, one of the notable early self-managing projects had been for military applications initiated by the DARPA[2] agency. In 1997 they started a project called Situational Awareness System[3] followed by the DASADA[4] project in 2000. The objectives were to research and develop technologies that would enable mission-critical systems to meet high-assurance, dependability, and adaptability requirements, essentially dealing with the complexity of large distributed software systems [Huebscher and McCann, 2008]. The DASADA project lead to pioneer an architecture-driven approach to self-management using a notion of monitors and adaptation engines for optimizing system operation based on observed data. The objectives and approach were close to what had been later proposed by the IBM's autonomic computing initiative (*cf.* Section 2.1.3).

2.1.1 Feedback Control Loop

The main principle behind any self-adaptation is in regulating system characteristics in response to a system or environmental state change. As the *self* prefix suggests, both the regulation and the state observation is done autonomously by the system itself with minimal or no human interventions. While there are different forms of self-adaptation mechanisms, in general, they involve some sort of *feedback* [Brun et al., 2009].

Feedback is generally defined as ``*information about the gap between the actual level and the reference level of a system parameter which is used to alter the gap in some way.*'' [Ramaprasad, 1983]. It means that an information is a feedback only if it is translated back into an action performed onto a system from which it has originated. The main idea of *feedback control loop* is to observe system's output properties, such as throughput or utilization and then to use them to alter the system parameters, such as queue size or scheduling policies in order to meet specified goals. Because the output properties are used to adjust the in-

[2] Defense Advanced Research Projects Agency: http://www.darpa.mil
[3] http://www.darpa.mil/ato/programs/suosas.htm
[4] http://www.rl.af.mil/tech/programs/dasada/program-overview.html

put (control) parameters, and these inputs then in some way should affect the outputs, the architecture is called *feedback control loop* or *closed loop* [Hellerstein et al., 2004]. An alternative to closed-loop feedback control is *open-loop control* sometimes referred as *feed-forward control*. It is a technique that does not observe the target system and uses only the reference input for the control [Hellerstein et al., 2004]. In this text we are concerned with engineering closed loop systems. The open loop systems can always be created by removing feedback.

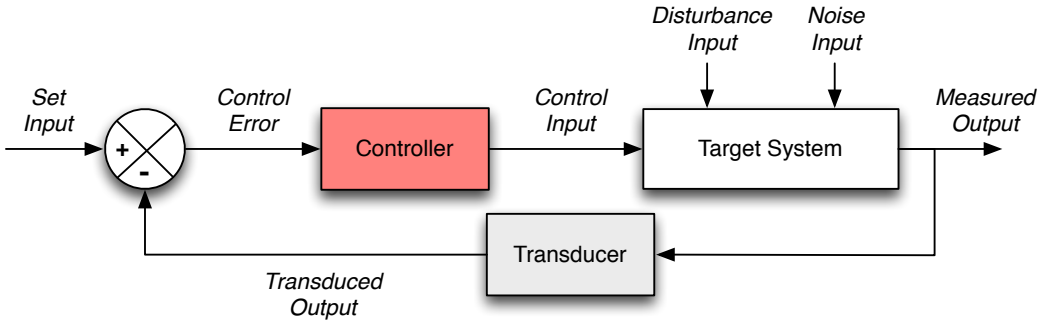


Figure 2.1: Conventional block diagram of a feedback control system. The *set input* is a reference value desired to be system's *measured output*. The controller adjusts the *control input* to align the system's output and the reference value. The *transducer* optionally filters (average, aggregate) the output values.

Figure 2.1 shows a classical *block diagram* of a single-input, single-output feedback control system. Block diagrams are used in control theory to model feedback or feedforward systems using components and signals that flow between them. A block in the diagram represents a system component (that can be the system itself) often labeled with the transfer function[5] it represents. The block diagram provides a useful and simple method for graphical analysis and reasoning about the control systems.

To illustrate the main principles of a feedback loop we take a canonical example of temperature control that is featured in many control engineering textbooks [Doyle et al., 1992, Hellerstein et al., 2004, Aström and Murray, 2009]. Consider an ordinary heating-control thermostat. This device measures the temperature (*measured output*) of a house (*environment*) and compares it with the desired set-point (*set input*). The feedback error between the two values (*control error*) is used to operate the house heating (*target system*), *i.e.*, to turn it on shall the measured temperature be too low or turn it off shall the temperature be too high. However, because of the lags and delays in the heater and the temperature sensor, the control model in the thermostat should do a bit of an anticipation, turning the heater on or off before the feedback error actually changes sign. Otherwise, the temperature would oscillate and there would be a lot of power-cycling in the heater, which on a longer run might cause a severe damage to the heating system.

Feedback systems have some very good properties. As pointed by Karl Aström the ``*magic of feedback*'' [Aström, 2006] is that it can create a system that performs well from

[5] A transfer function describes how an input is transformed into the output. For example, a transfer function of a controller defines how a control error is used to compute the control input.

components that performs poorly, make a system insensitive to disturbances and component variations, stabilize an unstable system and create desired behavior (*e.g.*, linear behavior from nonlinear components). On the other hand, as it has been shown in the above example, the major drawback of a feedback control is that it can cause instabilities.

2.1.2 Applications to Software Systems

In software systems, the feedback control is employed to steer the decisions that would be typically done at design-time, but instead, are moved towards runtime [Brun et al., 2009]. These decisions are therefore reassigned from software engineers and system administrators to the system itself [Andersson et al., 2012]. In other words, a key point of self-adaptive software is that its life-cycle continues after system deployment and that the system keeps evaluating its operational characteristics and responds to changes it observes at runtime [Salehie and Tahvildari, 2009].

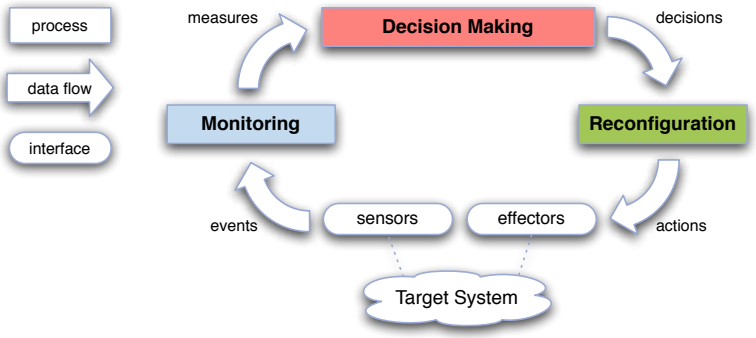


Figure 2.2: Conceptual feedback control loop in software systems

While it is conceptually the same, in the case of software systems the feedback loop is usually presented as a three or four steps process involving monitoring, decision-making and reconfiguration activities as depicted in Figure 2.2. The three main processes are:

- *Monitoring.* It is responsible for using sensors to collect all relevant measurements about the target system state. This may include any necessary data preprocessing such as filtering and stabilization mechanisms, conversions or rule inference.

- *Decision-making.* It essentially accounts for choosing an appropriate action among the set of all permissible actions. The actions are chosen based on the observed state of the target system following the system objectives. It is usually driven by some control model representing the structure and/or the behavior of the target system.

- *Reconfiguration.* It carries out the chosen action using the target system's effectors. This may be either a primitive action or a complete workflow of actions orchestrating the provided effectors in order to get the system into the desired state requested by the decision-making process.

Figure 2.2 shows a single feedback control loop. While it is obviously easier to design and reason about a single control loop, in practice, usually multiple separate loops are involved [Vromant et al., 2011]. In general, following the good engineering practice, the aim is to minimize the number of control loops or to decouple them in respect to time or space [Müller et al., 2009]. For example, one possibility is to compose loops in a hierarchical organization implementing the divide-and-conquer like concept. In this scheme, the loops at the higher levels influence the loops at lover levels, but usually operate at different time scales avoiding unexpected interference. This scheme is used in the autonomic computing reference architecture (*cf.* Section 2.1.3 on page 17). Next to this hierarchical organization, there exists a variety of different control schemes [Patikirikorala et al., 2012] with different levels of controller decoupling. What is important, however, is that the design should always make the loop interaction explicit and define how these interactions are handled. Any self-adaptive system architecture should therefore make the control loops explicit. Furthermore, it should to highlight some of the key aspects such as structural arrangements (*e.g.*, sequential, parallel, hierarchical, decentralized), interactions and data flow [Müller et al., 2009, Brun et al., 2009].

The feedback control loop in software systems is sometimes also referred to as *autonomic control loop* [Brun et al., 2009] or *adaptation loop* [Salehie and Tahvildari, 2009]. It is important to note, that there exist different conceptual representations of this loop. Literature following the IBM MAPE-K decomposition of a feedback loop (*cf.* Section 2.1.3 on page 17) includes an extra process between the monitoring and decision-making that is usually called *analyzing* or *detecting* being responsible for analyzing the symptoms from the monitoring process, detecting when a change is required. Others [Brun et al., 2009, Dobson et al., 2006] are using *Collect-Analyze-Decide-Act* loop process refining the AI community's *Sense-Plan-Act* approach from the 1980s [Gat, 1998, Nilsson, 1980]. Another studies [Bertran et al., 2012, Cassou et al., 2011] use the notion of *Sense / Compute / Control* (SCC) architectural pattern that is based on the work of Chen *et al.* [Chen and Kotz, 2002] and Taylor *et al.* [Taylor et al., 2010]. The SCC application pattern involves four layers: *sensors, context operators, control operators* and *actuators* that intuitively map to the feedback control loop introduced above. Similarly to [Ramirez and Cheng, 2010], in this thesis we use the three process representation as it is closer to the block schema from the control theory and we find it more generic and easier to use.

The monitoring and reconfiguration processes of the feedback control loop are built on the top of *sensors* and *effectors*. They are the manageability endpoints of the target system called *touchpoints* representing the interfaces introspecting the system state and exposing its management operations. They are an important part of any self-adaptive software system as the adjacent control can only be based on the information and management operations available from the connected sensors and effectors. Usually one of the first step in realizing self-adaptation in software is a clear identification of the accessible touchpoints.

From the separation of concerns perspective in respect to the control mechanisms and application logic, the adaptation can be divided into two categories: *internal* and *external*. The former one intertwines the application and adaptation functionality and embeds di-

rectly in the target system code base while the latter one encapsulates the control into an external adaptation engine running in parallel with the target system [Salehie and Tahvildari, 2009]. The main drawback of the internal adaptation is that it is usually hard-coded and hidden in the application code, and therefore difficult to analyze, verify, modify or extend. The external adaptation is on the other hand built separately, making it easier to reason about the control behavior. It separates the concerns of the target system functionality from that of adaptation. It allows one to built adaptation on top of legacy systems where the source might not be always available. Moreover, externalizing the adaptation potentially enables its further reuse across multiple adaption scenarios and systems. The main drawback of the external approach is however in assuming that the target system can provide (or can be instrumented to provide) all the necessary endpoints for its observation and consequent modification. This assumption seems reasonable since many of the systems already provide some interfaces (*e.g.* tools, services, APIs) for their observation and adjustment [Garlan et al., 2004]. Generally, externalizing the adaptation has a number of benefits and therefore in this thesis we are concerned with the external approach.

The conceptual model of the feedback control loop as introduced in this section provides an overview of a possible architecture for self-adaptive systems. However, because of its generality it serves merely as guidelines for high-level system implementations. Some details, especially those related to the decision-making part can be drawn on to control theory. Control theory uses feedback loop as its central element and provides well-established mathematical models, tools, and techniques for a systematic approach for designing control loops that are stable and accurate [Abdelzaher et al., 2008]. In particular, Hellerstein *et al.* [Hellerstein et al., 2004] provide a comprehensive overview about the capabilities offered by control theory and their application to computing systems. Furthermore, Shaw [Shaw, 1995] compares the suitability of control loops as a software engineering methodology stating that ``*when the execution of a software system is affected by external disturbances forces or events that are not directly visible to or controllable by the software this is an indication that a control paradigm should be considered for the software architecture*''.

The essence of a good control lies in defining a suitable model that enables to quantify the control objective. This is, however, far from being trivial. It requires a significant amount of domain-specific knowledge and involves an extensive experimenting with a variety of FCL procedures and tuning parameters. While some fields, *e.g.*, performance engineering and queuing theory already provide mature models, in many other application domains it is yet not the case and it is difficult to model the relationships between controlled inputs and outputs [Müller et al., 2009]. Moreover, there is a certain accessibility gap of control theory for computer practitioners, although the Hellerstein's book fills a great part of this gap.

2.1.3 Autonomic Computing

While the idea of self-adaptive software systems had not been new, the major breakthrough came with the IBM's autonomic computing initiative. Originating in the IBM's

manifesto [Horn, 2001] from 2001, the concept of *Autonomic Computing* (AC) found an inspiration in biological systems. It suggests to compare complex computing systems to the human body that is an extremely complex system in itself, but it has an *autonomic nervous system* (ANS). The ANS that takes care of most bodily functions (body temperature, heart rate, blood pressure, breathing rate, and many more involuntary reflexive responses), thereby removing from our consciousness the task of coordinating all our bodily functions. What is the most fundamental is that it does it autonomously, without any conscious recognition or effort in one's part allowing one to focus on goals (*what to do*) rather than on means (*how to do*). One of the best examples[6] of this is given in the manifesto [Horn, 2001]: ``*You can make a mad dash for the train without having to calculate how much faster to breathe and pump your heart, or if you'll need that little dose of adrenaline to make it through the doors before they close.*''

The later published architectural blueprint [IBM, 2006] proposes the first architecture for self-adaptive systems that explicitly exposes the feedback control loop [Brun et al., 2009]. The AC architecture aims at fundamental goals describing: the external interfaces and behaviors required of individual system components, how to compose these components so that they can cooperate toward the goals of system-wide self-management, and how to compose a system from these components in such a way that the system as a whole is self-managing [IBM, 2006].

As stated in the seminal paper by Kephart *et al.* [Kephart and Chess, 2003] the essence of AC is system self-management. The idea is that system administrators should become free from the low-level administration management and instead focus on defining higher-level goals for the autonomic system to follow. Yet, they should still remain in the loop by assisting and approving the self-management process.

The IBM's autonomic vision has been also followed by other initiatives from the computing industry [Müller et al., 2009] such as: Sun's N1 System Management Program to enable application development across heterogeneous environments in a consistent and virtualized manner[7], Microsoft's Dynamic Systems Initiative technology strategy for products and solutions to help businesses meet the demand of a rapidly changing and adaptable environment [Microsoft, 2004], Hewlett-Packard's approach to autonomic computing is reified in its Adaptive Enterprise Strategy [HP, 2003], or Intel with its involvement in the development of standards for autonomic computing [Tewari and Milekovic, 2006].

Autonomic Element and MAPE-K Autonomic Loop The IBM's blueprint also introduced the *Autonomic Computing Reference Architecture* (ACRA) with a notion of an *autonomic element* as a fundamental building block for designing self-adaptive systems. An autonomic element (Figure 2.3) consists of an *autonomic manager* and *managed resource*. An autonomic manager represents the controller and implements the self-managing behavior

[6] While the comparison with natural systems seems to light the vision, after a decade of Autonomic Computing, no work on applying ANS principles to autonomic computing has been undertaken [Kephart, 2011].

[7] After acquisition of Sun Microsystems by Oracle Corporation, the development of the N1 Service Provisioning has ceased. http://www.oracle.com/technetwork/documentation/legacy-ent-computing-193035.html

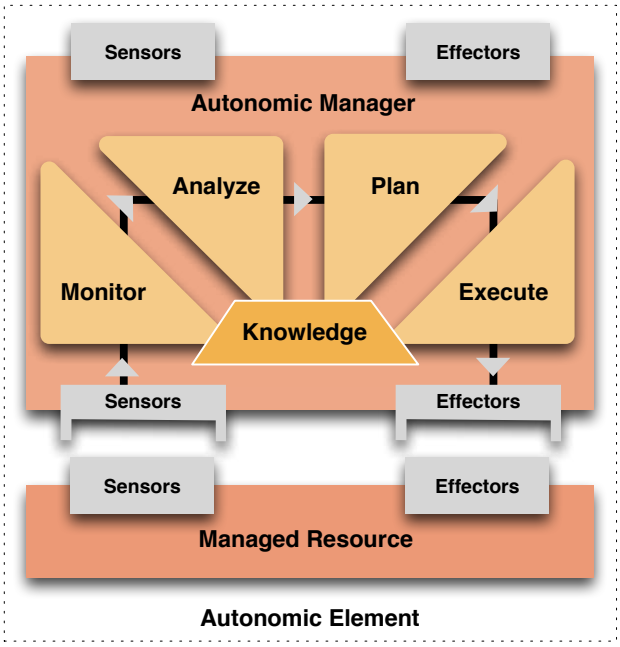


Figure 2.3: The IBM autonomic element with the MAPE-K reference model

using an autonomic control loop [IBM, 2006] which is often called the MAPE-K (*Monitor, Analyze, Plan, Execute, Knowledge*) loop. A managed resource represents the target system, *i.e.*, the running process, with two manageability interfaces: sensors and effectors.

The MAPE-K control loop shown in Figure 2.3 is conceptually similar to the feedback control loop from the classic control theory (*cf.* Section 2.1.2). It consists of four processes organized around a common *Knowledge* element representing the shared management data [IBM, 2006]:

- *Monitor* provides the mechanisms to collect, aggregate, filter and report monitoring data collected from a managed resource through sensors.
- *Analyze* provides the mechanisms that correlate and model complex situations and allow the autonomic manager to interpret the environment, predict future situations, and diagnose the current state of the system.
- *Plan* provides the mechanisms that construct the actions needed to achieve a certain goal, usually according to some guiding policy or strategy.
- *Execute* provides the mechanisms to control the execution of the plan over the managed resources by means of effectors.

The architecture follows the form of an external feedback loop. Moreover, an autonomic element itself can also be considered as a managed resource and it can also be controlled via the sensors and effectors that it exposes. This allows to build a hierarchical structure of feedback loops (*cf.* Section 2.1.2) and offers a separation of concerns regarding the overall system goals. The ACRA reference architecture defines a three-layer hierarchy

of orchestrating managers, resource managers, and managed resources which all share management data via a service interface [IBM, 2006]. The low-level autonomic managers near the managed resources control short term goals tuning directly parameters of the adjacent resources. The higher-level autonomic managers on the other hand participate in a more global view of the system and handle long term goals such as load balancing or higher-level modification of the system architecture. An autonomic element serves as the fundamental building block for realizing the self-* properties.

Self-* Properties Next to the concept of the autonomic element and MAPE-K loop, IBM has also distilled one of the initial set of the *self-** properties also known as adaptivity properties [Salehie and Tahvildari, 2009]. They encapsulate the eight main autonomic characteristics from the original manifesto [Horn, 2001] and have been defined in accordance to self-adaptation mechanisms found in nature [Kephart and Chess, 2003]. The main *self-** properties includes: (1) self-configuring, (2) self-healing, (3) self-optimizing and (4) self-protecting [IBM, 2006]. It is, however, mostly the self-optimization that has been targeted by the research community. In his ICAC'11 keynote *Autonomic Computing: The First Decade* [Kephart, 2011], Jeff Kephart states that the preponderance of work in the field continues to focus on self-optimization with the other *self-** properties receiving far less attention. The main reason is that it is easier to define benchmarks for self-optimization than for the other *self-** and ``*benchmarks drive the innovation.*''. According to Selehie and Tahvildari [Salehie and Tahvildari, 2005] the *self-** properties are linked to software quality metrics which could help define and understand these properties better and to utilize the existing body of knowledge on quality factors, metrics, and requirements, in developing and operating self-adaptive software.

Relation to Self-Adaptive Software Systems It is difficult to draw a real distinction between the self-adaptive software systems and the self-managing autonomic computing systems. In most of the related literature, we find that authors are using the terms self-adaptive (with or without adding software systems), autonomic computing, and self-managing interchangeably. In a recent survey by Salehie and Tahvildari [Salehie and Tahvildari, 2009], the authors tries to make some comparison concluding: ``*From one point of view, the self-adaptive software domain is more limited, while autonomic computing has emerged in a broader context. This means self-adaptive software has less coverage and falls under the umbrella of autonomic computing. From another point of view, we can consider a layered model for a software-intensive system that consists of: application(s), middleware, network, operating system, hardware [McKinley et al., 2004], and sub-layers of middleware [Schmidt, 2002]. According to this view, self-adaptive software primarily covers the application and the middleware layers, while its coverage fades in the layers below middleware. On the other hand, autonomic computing covers lower layers too, down to even the network and operating system... However, the concepts of these domains are strongly related and in many cases can be used interchangeably.*'' To the best of our knowledge there is no explicit comparison between these two and in this thesis we also consider them to be the same.

2.2 Related Work

There is a vast body of work on self-adaptive software systems and autonomic computing coming from different communities exploring the field from different perspectives. On the one hand this leads to a variety of innovations and overall understanding, but on the other hand, because the communities are rather loosely coupled, it also leads to the fragmentation of the field and to the development of different terminologies. Another difficulty in studying related work lies in the fact, that often it is difficult to infer some of the technical details and limitations of the presented work as they are not discussed[8].

In this section we give an overview of a selected work that is related to engineering of self-adaptive software systems and that we have identified as the most relevant to the context of this thesis. The study is organized into two parts. In the first part, we give a brief overview of the selected projects and techniques that aim at facilitating development of self-adaptive software systems (Section 2.2.1). This includes framework-based approaches, work enabling self-adaptation in the target system (*e.g.* dynamically adaptive middleware) and solutions for engineering parts of the adaptation engine (*e.g.* addressing monitoring or applying a particular decision-making technique). The intention is not to provide a exhaustive survey of engineering approaches of self-adaptive software systems, but rather to give an overview of the existing systems and techniques. Nevertheless, we believe that the included technologies and projects represent the most relevant approaches with respect to our aim. The selection has been mainly based on the following surveys:

- Huebscher and McCann, *A survey of autonomic computing: degrees, models, and applications* [Huebscher and McCann, 2008]
- Salehie and Tahvildari, *Self-adaptive software: Landscape and research challenges* [Salehie and Tahvildari, 2009]
- Cheng *et al.*, *Software engineering for self-adaptive systems: A research roadmap* [Cheng et al., 2009a]
- Villegasetal *et al.*, *A framework for evaluating quality-driven self-adaptive software systems* [Villegas et al., 2011]

In the second part, we provide more details about some of the projects (marked by † in the first part) that aim at providing a more generic approach for engineering complete self-adaptive software systems (Section 2.2.2). The projects were selected because (1) they show some of the representative frameworks, middleware and model-based including model@run.time approaches, and (2) they are close to our focus and had the most influence on our design.

[8] Weyns *et al.* [Weyns et al., 2012], surveys 96 publications from SEAMS and the associated Dagstuhl seminar between the years 2006 and 2011 reporting that the majority of the studies discusses no limitations and that in most cases assessment methods, tools and data are not publicly available.

2.2.1 Approaches Facilitating Self-Adaptation

The overview starts with framework-based approaches and solutions oriented towards adaptive middlewares. This is followed with an outline of model-based techniques and work related to feedback control and control theory.

Frameworks IBM proposed a form of standardized approach for autonomic systems using the MAPE-K decomposition and the ACRA architecture [Kephart and Chess, 2003, IBM, 2006] (*cf.* Section 2.1.3 on page 17). Following these general principles, a number of framework-based approaches have been developed focusing on different aspects of self-adaptation in software systems.

Litoiu *et al.* [Litoiu et al., 2005] describe a hierarchical framework that accommodates scopes and timescales of control actions, and different control algorithms. Their architecture considers three main types of controllers reflecting the three different stages that they focus on: tuning, load balancing, and provisioning. *Rainbow*† [Garlan et al., 2004] is a framework for developing architecture-based self-adaptation providing a reusable infrastructure (*cf.* page 24). *StarMX*† [Asadollahi et al., 2009] is designed for building self-managing Java-based applications using closed feedback control loops. It uses *Java Management Extension* (JMX) for target systems touchpoints and a policy-rule language or Java code for adaption engine implementation (*cf.* page 25). Similarly, *Autonomic Management Toolkit* (AMT) [Adamczyk et al., 2008] and *Adaptive Server Framework* (ASF) [Gorton et al., 2006] frameworks are employing rule engines for reasoning and decision making, and JMX touchpoints for building self-adaptive Java-based systems. A different approach enabling adaptable behavior in Java applications is followed by *TRAP* [Sadjadi et al., 2004], a framework that uses a combination of behavioral reflection and *Aspect-Oriented Programming* (AOP). Reflection and AOP techniques are also explored by Cámara *et al.* [Cámara et al., 2007] to develop a dynamic adaptation management framework that provides a basic infrastructure for a non-intrusive, semi-automatic approach of syntactical and behavioral adaptation.

Different approaches focuses on component adaptations. For example, *K-components* [Dowling and Cahill, 2004] introduces a component model enabling individual components self-adaption. They also include decentralized coordination models for collective adaptation of components groups based on machine learning techniques. *CASA* [Mukhija and Glinz, 2005] is a framework enabling dynamic application adaptation that supports various adaptation mechanisms including dynamic recomposition of application components. *JADE* [Bouchenak et al., 2005] presents a dynamically configurable component-based architecture based on a FRACTAL component model for autonomous repair management in distributed systems such as J2EE server clusters.

Some frameworks focus on managing the *Quality of Service* (QoS) in *Service Oriented Architectures* (SOA). For instance, *MOSES* [Cardellini et al., 2009], a methodology and a prototype tool, aims at driving the self-adaptation of a SOA system to fulfill non-functional QoS requirements such as system performance, reliability, and cost. *SCeSAME* [Tamura et al., 2010] proposes a component architecture for self-reconfiguration as an action asso-

ciated to QoS contracts violation. *FUSION* [Elkhodary et al., 2010] uses feature-oriented system models for learning impacts of adaption decisions on the system goals using reinforcement learning techniques. This knowledge is then used for automatic tuning of adaptation logic and runtime analysis.

Middlewares Next to frameworks, research in self-adaptive software systems has also focused on extending middlewares with self-adaptation capabilities to provide adaptation services separating applications from operating systems and network communications [McKinley et al., 2004]. These approaches aim at shielding developers from complex tasks such as resource distribution, component probing, network communication or application reconfiguration [Ramirez and Cheng, 2010]. On the other hand, middleware poses highly-specific execution environments which might not be directly applicable for some systems.

Océano [Appleby et al., 2001] is a *Service-Level Agreement* (SLA) based environment that aims at providing a highly available, scalable and manageable infrastructure. It can dynamically reallocate resources in response to changes in application load (*e.g.* allocation of servers for increased processing capacity or throttling network bandwidth when additional servers are unavailable) while preserving some security and isolation requirements. *Adaptive CORBA Template* [Sadjadi and McKinley, 2004] focuses on adaptive CORBA [Object Management Group, 2012a] applications including legacy systems without the need to change their source code. It transparently weaves adaptive behavior into object request brokers at runtime in response to changes in requirements and environmental conditions, supporting quality-of-service and fault tolerance. *M-Ware* [Kumar et al., 2007] provides a self-adaptive middleware solution that aligns automated system administration with business policies to maximize business utility in the domain of business information stream management. The adaptation is expressed using an enterprise environment model (*e.g.*, response-time, customer-priory) and a utility functions measuring the business value of a given environment state.

The use of self-adaptive mechanisms has been also actively studied in the domain of mobile and ubiquitous computing to prevail an uncertainty and unexpected changes in the target system execution context. The *MADAM* [Floch et al., 2006] project aims at facilitating development of adaptive applications for mobile computing. It employs explicit architecture-centric approach using models for information including structural and distribution aspects together with utility functions for comparing their variability. *CARISMA* [Capra et al., 2003] is a middleware for mobile computing exploiting the principle of reflection to facilitate construction of adaptive context-aware interactions between mobile applications. The adaptation is described using policies based on microeconomic sealed-bid auction approach. *MUSIC* [Rouvoy et al., 2008] is a QoS-driven generic middleware for self-adaptive mobile applications for ubiquitous environments where the availability and the quality of offered remote services change. It exploits these changes by discovering, evaluating and binding remote services available in the surroundings of mobile devices. Similarly, *CAPPUCINO*† [Romero et al., 2010] is also based on a device-centric

utility-based approach to support context-aware Web Services execution in ubiquitous environments (*cf.* page 27).

Model-Based Approaches Zhang and Cheng [Zhang and Cheng, 2006] introduce an approach to create formal models for behavior of adaptive programs, to automatically analyze them, and generate their implementations. Their approach separates adaptation behavior and non-adaptive behavior specifications thereby making the models easier to specify, inspect and modify. Further, Ramirez and Cheng [Ramirez and Cheng, 2010] study adaptation-oriented design patterns that support the development of adaptive systems. They harvest 12 designs that support different FCL phases and that facilitate the separate development of the functional logic and the adaptive logic.

The use of model as a formal specification of self-adaptive software systems is also proposed by Weyns *et al.* [Weyns and Malek, 2010] using their *FORMS* reference model. *FORMS* is a formal reference model based on the MAPE-K (*cf.* Section 2.1.3 on page 17) supporting composition of adaptation mechanisms, allowing one to capture their key characteristics and compare alternative solutions. Another reference model for engineering self-adaptation is *DYNAMICO* [Villegas et al., 2013]. It is based on a three-layer architecture defining three types of FCL, each managing different parts of context dynamics (control objectives, target system adaptation and dynamic monitoring). Hebig *et al.* [Hebig et al., 2010] proposes a UML profile for explicit architecture of several coordinated control loops using component diagrams allowing to specify interactions between them.

Several approaches are exploiting the use of model-driven engineering techniques for self-adaptive software systems. *Genie* [Bencomo et al., 2008] uses models for specification and generation of middleware-based software artefacts. It utilizes architectural models to support generation and execution of adaptive systems for component-based middleware technologies. The adaptive logic is specified as state machines, with each state being a system configuration and transitions being reconfiguration scripts. *J2EEML* [White et al., 2005] relies on models for describing design of *Enterprise Java Beans* (EJB) applications[9], their QoS requirements, and autonomic properties. This is accompanied with tools for code generation, automatic checking of model correctness, and visualization of complex QoS and autonomic properties. *Diasuite*† [Bertran et al., 2012] is a tool suite based on generative programming to engineer SCC applications, providing support that covers the whole SCC development process (*cf.* page 28). *Ptolemy*† [Eker et al., 2003] is a modeling environment implementing an actor-oriented design methodology and hierarchical heterogeneity suitable for designing and simulating control systems (*cf.* page 29).

A different model-based approach to self-adaptive software engineering is based on the idea of using models and model-driven engineering techniques at runtime instead of at design time. The *model@run.time* or *runtime model* represents an abstraction of a running system or its part and can be used to support dynamic adaptation of structure, behavior or goals of the underlying software systems [France and Rumpe, 2007]. This abstraction is causally connected with a given system during its actual execution. The causal connection

[9] http://www.oracle.com/technetwork/java/javaee/ejb/index.html

allows the model to provide up-to-date information about the state of the system and enables the adaptations to be done at the model level rather than at the system level [Blair et al., 2009]. For example, DiVA [Morin et al., 2009] and the following Kevoree [Fouquet et al., 2012a] projects use architectural models at runtime for dynamic adaptation, together with modeling some other aspects such as variability, system context and adaptation logic. It also employs MDE techniques to produce reconfiguration scripts that perform the actual adaptation. Another approach in model@run.time is taken by Vogel and Giese [Vogel and Giese, 2012] providing domain-specific modeling language for representing software systems and adaptation mechanisms using *Executable Runtime Megamodels*† (*cf.* page 31).

Feedback Control and Control Theory There has also been a considerable effort in feedback control and control theory application for software systems. We have already mentioned the work by Hellerstein *et al.* [Hellerstein et al., 2004] that provides a concrete approach for designing feedback control mechanism for computing system based on control theory.

There have been many studies employing a particular control technique. Patikirikorala *et al.* [Patikirikorala et al., 2012] conducted an extensive survey of 158 papers using control engineering approaches for designing self-adaptive software systems. It shows that the use of control theory is indeed very relevant and presents a classification of the approaches harvesting some common design patterns used for controller design. Among the studies within the survey, we mention Abdelzaher *et al.* [Abdelzaher and Bhatti, 1999, Abdelzaher et al., 2002] work on *Quality of Service* (QoS) management of web servers using adaptive content delivery. We reuse their scenario in this thesis to illustrate our approach on a real adaptation scenario (*cf.* Section 3.2). Many studies such as this one focus on sophisticated models of adaptation engines leaving the integration into target systems to some low-level system programming, usually without any systematic approach.

Maggio *et al.* [Maggio et al., 2011] present and compare different state-of-the-art decision making approaches. Furthermore, in her PhD thesis [Maggio, 2011] the use of control theory principles to drive the design of computing system is discussed.

2.2.2 Approaches Aiming at Generic Self-Adaptation

In this section follows a more detailed overview about some of the approaches we find the more relevant to our thesis.

Rainbow One of the first frameworks facilitating the engineering of self-adaptive software systems is Rainbow [Garlan et al., 2004]. In its core, it is composed as a two-layer framework (*cf.* Figure 2.4) with an external fixed control loop supporting architectural-based system representation and adaptation strategy. The adaptation process is based on the use of a customized Acme-based [Garlan et al., 2000] software architectural model at run-time to monitor and adapt target systems. The notion of architectural style has been augmented with the notions of operators (to change an architecture) and adaptation strategies (to group changes for specific purpose). The decision-making part is based on

the utility theory that considers multiple factors while being sensitive to the context of use.

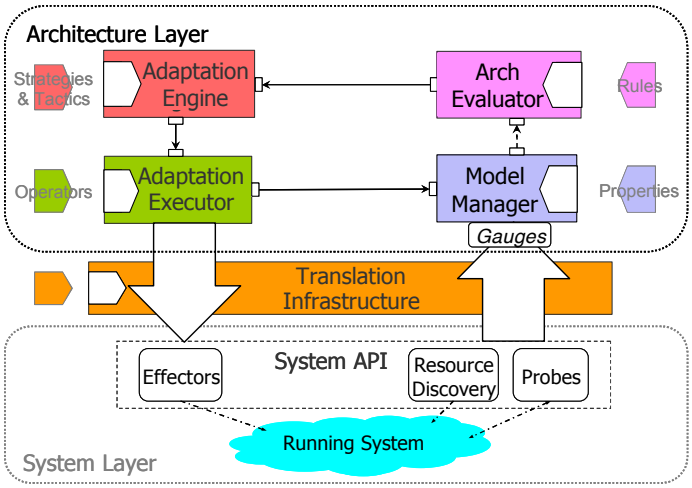


Figure 2.4: Rainbow Framework [Garlan et al., 2004]

The self-adaptation cycle in the Rainbow framework is realized using a MAPE-K loop (*cf.* Section 2.1.3 on page 17): monitoring is achieved by *Probes and Gauges* updating the model; analysis is performed by the *Architecture Evaluator* assessing problems on the model; decision occurs through the *Adaptation Manager* choosing actions based on the model states; action is accomplished by the *Strategy Executor* effecting changes on the system via Effectors; and knowledge is represented in the architecture model by the *Model Manager*.

Next to the core framework, Rainbow also provides a DSL, *Stitch*, for capturing explicit adaptation policies. The policies are expressed using the first-class entities adaptation concepts of an *operator* to capture a system-provided command, a *tactic* to describe a single adaptation step with cost and benefit impact including a condition statement in form of a structural invariant, and a *strategy* which is a packaged pattern of adaptation steps. Rainbow also comes with an integrated development environment, RAIDE, that enables editing the Acme architecture models, write Stitch scripts, simulate adaptation behavior, and deploy the Rainbow runtime [Cheng et al., 2009c].

While the feedback control loop is made explicit, the framework itself has several limitations. It has not been designed to allow runtime adaptation of this loop and the adaptation processes are fixed during the system execution. The framework is not explicitly reflective and therefore it does not support constructing different forms of hierarchically organized FCLs. The use of utility theory is one of the major underpinning of this approach and therefore it is hardwired into both the core framework and the DSL.

StarMX Following the IBM autonomic element architecture (*cf.* Section 2.1.3 on page 17), StarMX [Asadollahi et al., 2009] is a framework for building external self-adaptive Java-

based systems. It uses JMX technology[10] to enable self-adaptive capabilities in the target system. The JMX *mbeans* are used to represent the manageability endpoints of the target system. The decision-making logic is implemented as a set of entities, called processes, that form an autonomic manager (*cf.* Figure 2.5). Each process supports one or more parts of the MAPE-K control loop decomposition. It allows developers to implement the decision-making process in plain Java as well as in a higher-level rule based language *Common Information Model-Simplified Policy Language* CIM-SPL [DMTF, 2007] using Apache Imperius[11].

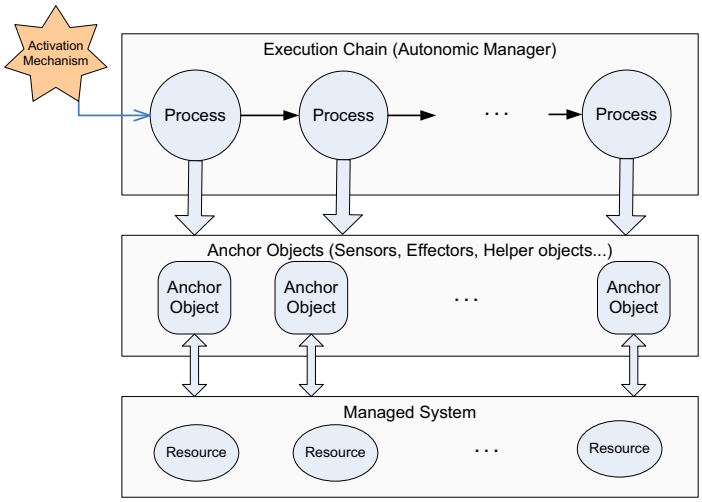


Figure 2.5: StarMX Framework [Asadollahi et al., 2009]

Each autonomic manager is associated with an activation mechanism that when activated sequentially executes the processes chain. The two activation methods supported by StarMX are: *timer-based* when the execution chain is triggered at fixed time intervals (*e.g.*, every 10 seconds), and *event-based* when the execution chain subscribes to receive a particular event from the managed system or from other sources like the StarMX processes. The communication within the framework is performed through a set of services that are responsible for operations such as locating the declared MBeans, initiating the activation mechanism, defining the scope of the actions, data sharing among processes, and obtaining logging information. The framework is using an XML configuration file to describe the assembly of the JMX MBeans, the processes and the execution chains, and to define the activation mechanisms.

StarMX is by the choice of technology primarily focused on adapting Java-based systems, although, there is a possibility to wrap other interfaces into JMX MBeans. The framework is, however, static. The manageability endpoints and system properties are defined in a configuration file that are not amenable at runtime (*e.g.* the timer-based ex-

[10] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[11] As of May 2011, the work on Apache Imperius has been stopped and the project is retired https://whimsy.apache.org/board/minutes/Incubator.html. Former project homepage: http://incubator.apache.org/imperius/

ecution period is fixed). Neither does it allow runtime modification of the management logic nor hierarchical composition of the autonomic managers. Moreover, it is also not clear what shall happen if, for example, the execution of an adapted process takes longer than expected (*e.g.*, due to a network delay) and in the mean time the very same process is fired again.

CAPPUCINO To tame the challenges mobile applications have to face in ubiquitous environments, CAPPUCINO [Romero et al., 2010] proposes a uniform approach to context-aware end-to-end adaptation by exploiting the *Service Component Architecture* (SCA) [OASIS, 2007] technology. The platform is built on the top of the FraSCAti and its lightweight version FraSCAtiME for mobile clients [Seinturier et al., 2009]. The control loops are implemented as distributed COSMOS [Conan et al., 2007] context policies using SPACES [Romero et al., 2010] for mediation via RESTful services.

– *FraSCAti* is an SCA implementation and the underlying execution engine for CAPPUCINO. Unlike standard SCA implementations, FraSCAti embeds the FScript engine [David et al., 2009] into SCA component thus providing the platform with reconfiguration capabilities.

– *COSMOS* is a component framework for collecting and composing context data for context-aware applications. It uses a hierarchy of context nodes representing a software component that encapsulates context information. It is a three-layered framework consisting of: *collectors* gathering raw data about the environment, *processors* filtering and aggregating raw data coming from context collectors, and *adaptation* representing the decision-making process. A context node also defines a set of properties that specifies its behavior: active / passive (associating thread control), observation / notification (communication type with respect to the node's hierarchy), and pass-through / blocking (propagating or blocking communication).

– *SPACES* enriches COSMOS with lightweight communication protocols and alternative resource representations on the top of *REpresentational State Transfer* (REST) [Fielding, 2000] architecture, provisioning *context as a service*. The COSMOS context nodes are published as REST resources and SPACES acts as a mediator being responsible for disseminating the context information among the entities.

Figure 2.6 shows an example of a deployment of the CAPPUCINO distributed runtime on tree distinct nodes. The autonomic control loop is implemented as an SCA application adapting mobile clients and application servers as they join the adaptation domain. The managed applications and clients use SPACES to connect the remotely deployed CAPPUCINO elements. The *Adaptation Runtime* is responsible for executing the autonomic loop. The collected context information is processed by the *Adaption Policy* so that the *Decision Engine*, and the *Reconfiguration Engine* components can create corresponding reconfiguration plans. These are then sent to their counterparts located in each distributed location, completing the loop.

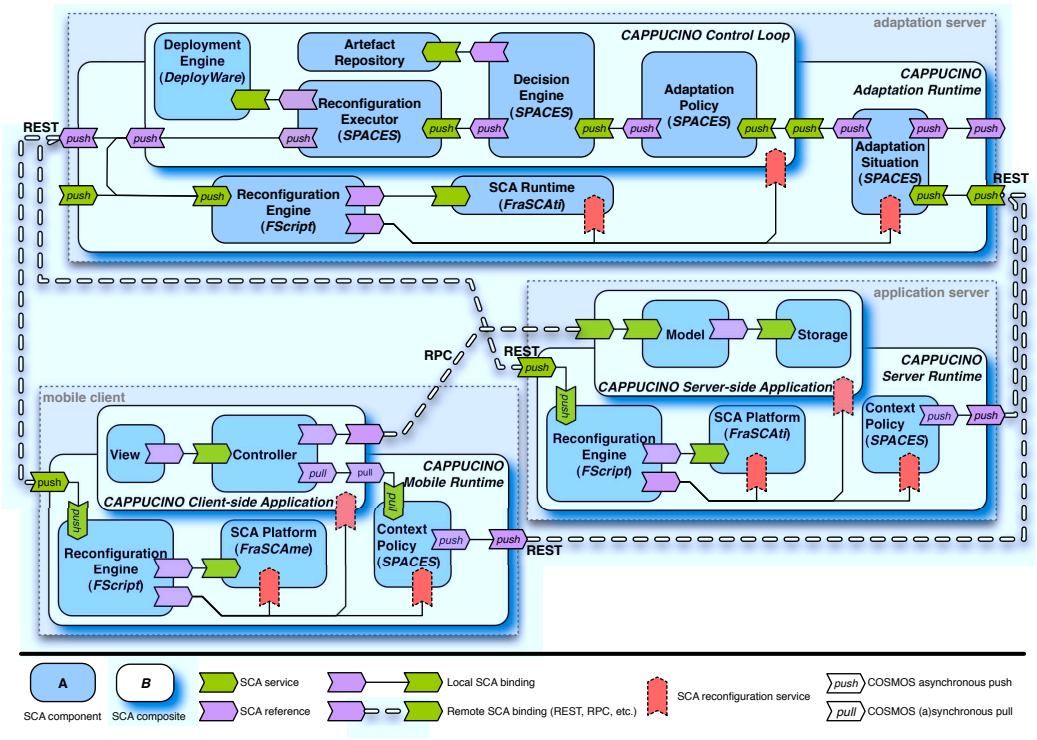


Figure 2.6: CAPPUCINO Framework [Romero et al., 2010]

CAPPUCINO primarily aims at adapting mobile clients and applications that are based on the SCA architecture. Since the control loop processes are themselves implemented as SCA components, they could also be reconfigured using FScript and therefore allowing hierarchical organization of multiple control loops. There is however no explicit multiple loop coordination support. Furthermore, SCA components can be executed simultaneously multiple times by multiple threads and thus it is upon developers to make sure that service implementations are thread safe. Finally, architecture descriptions are kept in SCA composites imposing the use of SCA technologies limiting the target deployments to the availability of SCA runtime.

DiaSuite Focusing on the Sense/Compute/Control (SCC) applications, DiaSuite [Bertran et al., 2012] is a tool suite supporting their development using a software design approach. The tool suite contains a domain-specific design language, a compiler producing a Java programming framework, a 2D-renderer to simulate an application, and a deployment framework. While the main focus is on ubiquitous application domains such as telecommunications or building automation, it can also be used for other systems such as web server monitoring (*cf.* Figure 2.7) [Cassou et al., 2011].

An application in the SCC pattern is decomposed into four layers (*cf.* Figure 2.7): *sensor*, *context operator*, *control operator*, and *actuator*. The behavior of the application is specified as an information data flow using an oriented graph where nodes represent the components and edges the data exchanges between them. Each layer contains a different

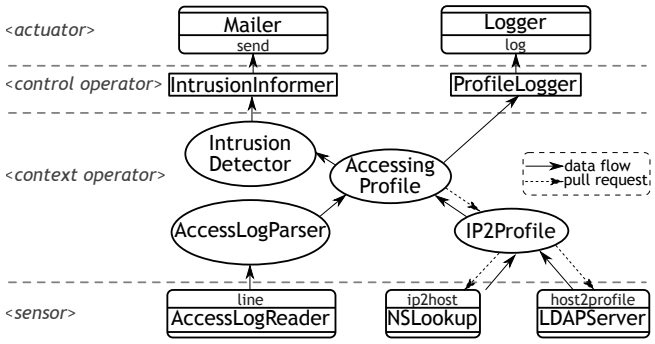


Figure 2.7: DiaSuite Example [Bertran et al., 2012]

class of components: *sensors* retrieve information from the environment; *context operators* interpret these information and either push data to other context operators and to control operators, or respond to requests from parent context operators; *control operators* translate the information coming from the context operators into orders for the actuators; and finally *actuators* trigger actions on the environment, finishing the conceptual autonomic feedback control loop.

DiaSuite enriches the SCC application pattern with a notion of *interaction contracts* to describe allowed interactions among the components. An interaction contract specifies the condition upon which a component is activated, what data requirements it might have (*i.e.* what additional data it might need to fetch in order to produce a result) and to what connected parents will it push computed result. For example, `AccessingProfile` from Figure 2.7 has a following interaction contract:

$$\langle \Uparrow (\texttt{AccessLogParser}); \Downarrow (\texttt{IP2Profile}); \Uparrow \texttt{self} \rangle$$

It indicates that `AccessingProfile` is triggered by a push request from the `AccessLogParser`, during its execution it might pull data from `IP2Profile` and it will always push the results to all its connected parents, in this case to `IntrusionDetector` and `ProfileLogger` operators.

Similarly to StarMX and Rainbow, the model is not explicitly reflective and does not allow hierarchical control composition neither it allows dynamic reconfiguration of the running platform.

Ptolemy There is a large body of work that concerns designing feedback control for embedded computing [Arzen and Cervin, 2005]. One of the prominent examples is the Ptolemy project [Eker et al., 2003] for modeling, simulation, and design of concurrent, real-time, embedded systems with a focus on concurrent components assembly. Ptolemy is an extensive framework for simulation of concurrent actor-oriented systems, with the ability to combine heterogeneous models of computation that govern the interactions between the actors. It allows for exploring interactions using models of dataflow and process net-

works, discrete-event systems, synchronous / reactive languages, finite-state machines, communicating sequential processes, and the like.

Among others, one of the novelty of Ptolemy is that the semantics of a model is not determined by the framework, but instead it is encapsulated in a software component (a director), that is added to the model. An actor in the model can contain other actors thus forming composites with arbitrary nesting (*cf. CompositeActor* in Figure 2.8). Each composite specifies its director that controls the execution of the actors within the composite thus allowing the heterogeneity (*e.g. Director 1* uses synchronized data flow model while *Director 2* is based on continuous time). The concept of a model computation is referred to as a *domain* and directors are associated with domains. Most actors are domain-polymorphic (agnostic about how their inputs and outputs are received and sent) and thus usable in multiple domain.

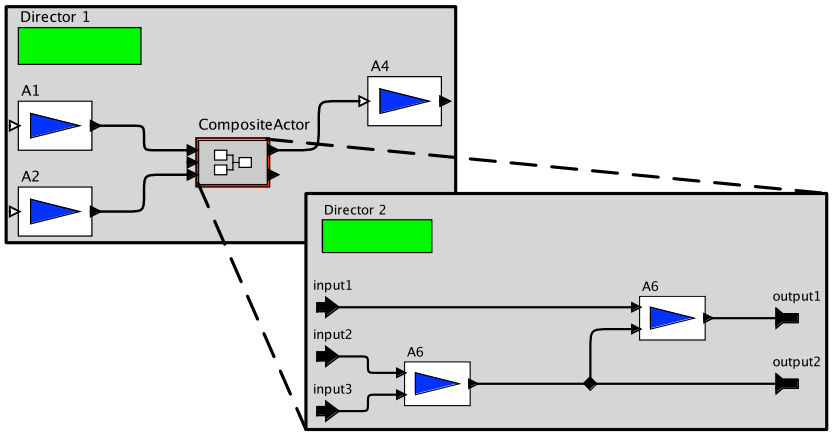


Figure 2.8: Example of a Ptolemy II model

One of the directors included in the current version of Ptolemy is *Component Interaction* (CI) [Zhao, 2003][12]. It focuses at data-driven and demand-driven styles of computation, which is a common communication style between software components, also referred to as *Push-Pull* communication. The push indicates a message transfer initiated by the producer while pull represents message transfer that is initiated by the consumer.

Ptolemy provides a state-of-the-art environment for designing and simulating control system with a particular focus on embedded control. It has been successfully used for designing complex heterogeneous control systems such as discrete computer based systems interacting with continuous physical plants. To the best of our knowledge, we are not aware of the usage of Ptolemy for building self-adaptive software systems. One of the current limitation is that adding a new actor into Ptolemy is not a straight-forward process[13]. Moreover, the use the codegen to synthesize executable systems from a model definition might not be suitable for the class of software systems we aim at. The codegen is limited to C and VHDL code with the main focus on embedded systems [Leung et al.,

[12] The Component Interaction model of computation is experimental and is included only in the full Ptolemy II: http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/ptII8.0.1/doc/domains.htm

[13] http://chess.eecs.berkeley.edu/ptexternal/nightly/doc/coding/addinganactor.htm

2008]. Also, currently only the synchronous dataflow and the heterochronous dataflow[14] can be used and it requires to associate each used Ptolemy actor with an adapter class that defines the C code implementation.

Executable Runtime Megamodels Following the model@run.time approach, Executable Runtime Megamodels [Vogel and Giese, 2012] provide a domain-specific modeling language for specifying and executing external feedback control loops. Megamodels are essentially models that have other models as their elements and that capture the relationships between the models by using model operations such as model transformations [Bézivin et al., 2005]. The executable runtime megamodels are using models to represent the target system and model operations for capturing the adaptation steps. The provided modeling language resembles a flowchart and data flow diagram (*cf.* Figure 2.9). The model operations are usually implemented using some model-driven engineering techniques such as model validation or transformation. For example a model operation can be an OCL invariant checking some structural constraint of the model representing the target system. The target runtime system is synchronized using triple graph grammar rules with an architectural model that is then used as the base for modeling the feedback control processes. The different models used within megamodels are stereotyped in order to differentiate the feedback loop concepts following the MAPE-K decomposition (*cf.* Section 2.1.3 on page 17). At runtime, these models are executed by an interpreter that follows the specified control flow and invokes the model operations.

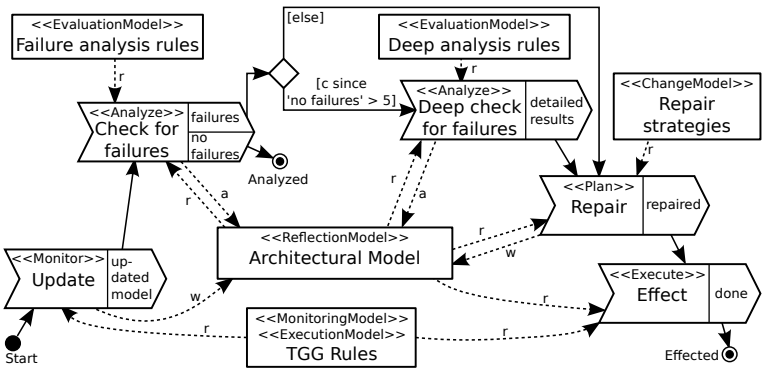


Figure 2.9: Executable Runtime Megamodels Example [Vogel and Giese, 2012]

Besides being able to express the adaptation concerns explicitly using a high level of abstraction, executable runtime megamodels provide support for multiple loop coordination with explicit synchronization. Since the models are reflective, they also support hierarchical organization of multiple loops operating on different time scales. The loops can be organized into layers where each layer is responsible for managing an adjacent layer below and the bottom most layer is the target system. The reflection capability allows for example to change the model operations or the control flow.

[14] http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.1/ptII8.0.1/doc/domains.htm#hdf

While the megamodels are accompanied with a runtime interpreter there is only a high-level overview of how the actual adaptations look like. Details about models data and control flow synchronizations, operations scheduling and timeliness are not provided. Furthermore, there might be some performance limitations of the approach since the megamodels are based on EMF that was primarily developed to be used at design-time. It has therefore some shortcomings for the use at runtime such as higher memory footprint and lack of thread safe access to the model elements [Fouquet et al., 2012b]. Similarly, many of the techniques for EMF model manipulation (*cf.* Section 6.2) primarily focus on design time.

2.3 Summary

In this chapter we have presented the context of self-adaptive software systems and made an overview of some of the representative approaches that facilitate their engineering. While the given selection is not exhaustive, it shows the diversity of approaches and different perspectives including conceptual and implementation-specific methods for tackling the engineering issues. In addition, we observe that despite this diversity there is no agreed-upon solution that would be preferred over the others. In many cases, the proposed approaches are designed for a particular problem, pointing out a problem-specific solution (*e.g.*, using a specific control algorithm or enabling adaptation for a particular system platform). Usually these solutions are tightly coupled with specific domains or technologies or depend on a particular execution environment, thus limiting their applicability with respect to the problem being addressed [Ramirez and Cheng, 2010]. Furthermore, while there have been lot of advances in mechanisms to enable self-adaptation, not much effort has been put into providing a systematic approach facilitating the integration of these mechanisms from an end-to-end system perspective. Consequently, different system parts are expressed at different levels of abstraction. This results in reduced visibility of the control mechanisms making it difficult to reason about them and to trace the higher level adaptation concepts from the low level implementation code. This limitation affects their reusability and flexibility making them difficult to apply in practice. Consequently, this gives raise to accidental complexities that contribute to increased cost of engineering and maintaining these systems.

This issue has been already identified and acknowledged [Cheng et al., 2009a, Brun et al., 2009, Salehie and Tahvildari, 2009]. According to us it is well summarized by Simon Patarin and Mesaac Makpangou in their *System-Level Support* [Babaoglu et al., 2005] in which they argue that it is a good system-level support that is missing for engineering self-adaptive software systems: ``*One that would present the right abstractions to the developers, which would allow them to prototype autonomic applications rapidly. One that would be flexible enough to cope with the diversity and the heterogeneity of current platforms. One that would be efficient in terms of performance, because you simply cannot pretend maintaining any sort of quality of service if the service you propose is of poor quality right from the beginning. And one that would keep simple applications easy to implement. It is our belief that such a basis would allow*

researchers with different motivations and experiences to put their ideas in practice, free from the painful details of low-level system implementation.''

Therefore we argue that there is still a need for an integrated approach for developing self-adaptive software systems. An approach that would provide a flexible feedback control loop abstraction with good system-level support allowing to integrate variety of self-adaptation mechanisms into a wide range of software systems.

In the next chapter, we will revisit some of the associated challenges and introduce our model-driven engineering approach for developing and integrating self-adaptive software systems.

CHAPTER 3

Modeling Feedback Control Architectures - Syntax

In the previous chapter we have presented an overview of the landscape of self-adaptive software systems. The remaining chapters of the first part of this thesis present our model-driven engineering approach that facilitates integration of self-adaptive mechanisms into software systems. In this chapter we introduce our approach and present a domain-specific model language for defining amenable feedback control architectures for external self-adaptive software systems based on feedback control loops.

We start with a discussion of the main design decisions. Next, we introduce a running example that will be used throughout the rest of this thesis to illustrate our approach on a concrete self-adaptation scenario. The rest of the chapter presents the main concepts of the language.

The language syntax is followed by a description of its semantics (Chapter 4) and an overview of a modeling environment and tools (Chapter 5) that facilitates the usage of the language.

3.1 Design Decisions

In the introduction section (on page 2) we have split the issues of self-adaptive software engineering into concerns related to (1) designing the underlying adaptable software system, (2) designing the adaptation engine, and (3) forming the architecture connecting them together. At the end of the last chapter (*cf.* Section 2.3), we have discussed the lack of support for the last one making the case for the need to provide a more integrated approach. In this thesis we focus on this issues, *i.e.*, on integrating the self-adaptive mechanisms into software systems by using feedback control loops.

In this section we revisit some of the challenges related to our integration objective and elaborate the main design decisions and principles followed in this thesis.

3.1.1 Challenges Revisited

Extracting from challenges identified in previous studies, in particular by Brun *et al.* [Brun et al., 2009] and Selehie and Tahvildari [Salehie and Tahvildari, 2009], we identify the following challenges for our work:

- *Generality.* Make the solution both *domain-agnostic* and *technology-agnostic* [Cheng, 2008, Salehie and Tahvildari, 2009]. Domain-agnostic so it is applicable to a wide range of software systems and adaptation properties. Technology-agnostic so it does not impose any particular technical solution for the target system nor for any of the feedback control loop processes.

- *Visibility.* Make feedback control loops, their processes and interactions explicit at design time and at runtime enabling coordination of multiple control loops using different control schemes [Müller et al., 2008, Brun et al., 2009]. This should increase the overall understanding of the self-adaptive capabilities and ensure a strong mapping between design and runtime control concepts.

- *Tooling.* Provide tool support that would aid developers in engineering of these systems and allow them to automate some recurring development tasks involving design, implementation and analysis of FCL [Brun et al., 2009, Salehie and Tahvildari, 2009]. There should be a support for traceability from the control design to the runtime implementation and some validation and verification techniques to assist in evaluating the correctness and maintenance of the control system.

- *Remote distribution.* Software systems are becoming increasingly networked and as a result, feedback control might involve remote, potentially geographically distributed applications. In other cases, because of practical or technological issues, it is either not desired or not possible to run the complete adaptation engine next to the target system and only remote touchpoints are available. Therefore, it is necessary that the approach supports remote distribution of feedback control processes without requiring lot of engineering effort.

Furthermore, the approach should follow good software engineering practices and allow modular specification of the self-adaptive behavior with the possibility to compose and reuse already implemented parts of the feedback control across multiple scenarios. Finally, in order to improve productivity and reduce the cost of engineering and evolving self-adaptive software systems, the approach should remain *lightweight*. It should facilitate engineering of such systems without requiring a lot of development effort making it easy to use without raising accidental complexities. A particular emphasis should be put on making the process automated where possible.

One way to address these challenges is to raise the level of abstraction at which these systems are being described. This will make it possible to reify feedback control loops as first class entities, improving the ability for reasoning on control mechanisms, make them amenable for analysis and synthesis of its implementation. In the rest of this section, we

will motivate the use of *Model-Driven Engineering* (MDE) techniques as a viable solution to meet the identified challenges and requirements.

3.1.2 Why a Model-Driven Engineering Approach?

At a very high level, a software development process can be seen as a function mapping between a *problem space* represented by domain-specific[1] abstractions and a *solution space* that consists of implementation-oriented abstractions [Czarnecki, 2005]. In the traditional development, this mapping is done manually through the process of programming. A developer takes requirements usually defined by a natural language using the abstractions from the problem space and maps them to the implementation abstraction offered by some *General Purpose Programming Language* (GPL). One of the problem with this process is the misalignment between the abstractions, *i.e.*, between the problems being addressed and their actual software implementations [Schmidt, 2006]. The conceptual gap between the problem and solution domains gives raise to the accidental complexities making the development difficult and costly [France and Rumpe, 2007]. In order to improve productivity (not just the speed, but also the quality), software engineers seek ways to raise abstractions used in the solution space [Kelly and Tolvanen, 2008].

Frameworks A software framework provide abstractions with extensible and reusable building block for some of the common and recurring problems of an application construction in a particular application domain [Johnson and Foote, 1988]. It gives an architectural basis for an application, defining its overall structure and taking responsibility over its control [Gamma et al., 1994]. In the Section 2.2 of related work, we have discussed a number of studies that base there solution on some sort of a framework.

The main advantage of a framework is that it can simplify development. It uses the principle of inversion of control [Gamma et al., 1994], where the user code gets called by the framework, allowing developers to concentrate on the specificities of their problems with the application domain and thereby enabling them to build the applications more rapidly.

On the other hand, frameworks operate within boundaries of some programming language (usually a GPL). Therefore they are limited in the level of abstraction they can provide. The possibility of a formal reasoning and verification is rather limited since the structure and behavior is an integral part of the implementation. Their implementation always imposes the use of certain technologies that might not be suitable for certain problems. Moreover, some creativity freedom is lost since many design decisions have been already made by the framework designers [Gamma et al., 1994].

Models A different approach is followed in the area of model-driven engineering, where the software development artifacts are models representing abstractions of some aspects of a system [France and Rumpe, 2007]. When these models are created using the problem

[1] A *domain* in this context represents ``*an area of interest to a particular development effort.*'' [Kelly and Tolvanen, 2008]

domain concepts, we talk about *Domain-Specific Modeling* (DSM). One of the advantages of using DSM is that it can significantly raise the level of abstraction as it can specify solutions directly using problem-level abstractions [Kelly and Tolvanen, 2008]. The solution, *i.e.*, system structure and behavior, is therefore separated from its implementation as it is captured at a conceptual level using the problem domain concepts, rather than the implementation concepts. Domain-specific models are usually described in *Domain-Specific Modeling Languages* (DSML), often specified using meta-models. A DSML defines a structure, a behavior and relationships among the problem domain concepts including their semantics and associated constraints [Schmidt, 2006]. Next to DSML, DSM includes a *domain-specific code generator* that uses the models to synthesize concrete software implementations of running systems. The code generator is usually coupled with a *domain framework* that provides a well defined set of services for the generated code to interface with [Tolvanen and Kelly, 2005].

This concept of software development is often referred to as *Model-Driven Software Development* (MDSD), a MDE branch that is concerned with generating code from models. It is important to realize that automation of the software synthesis is possible because both the language and the generator operates only within a certain, usually limited, domain (as opposed to general purpose models and languages). Also, while MDSD uses a framework, its use is separated from the solution as opposed to the case when the framework with the user code is the only representation of the solution. The code generator is an example of a broader concept called *model transformation*, a process whereby one or more source models are transformed into one or more target models, usually of a different purpose (model-to-model transformation) or to text (model-to-text transformation).

A note on UML *Unified Modeling Language* (UML) [Object Management Group, 2011b] is a general purpose modeling language that can be used for expressing wide range of systems and domains. However, we find two problems in using UML in our approach: (1) the size and complexity of UML and UML profiles, and (2) the generality that makes it more specific about implementation choices and thus limits the abstraction that can be expressed. While the use of UML profiles can make UML more domain-specific, still it cannot hide the underlying complexity of the UML models [France et al., 2006]. For example, the closes standard profile to our domain is the *SysML* profile [Object Management Group, 2012c], which aims at providing general-purpose modeling for system engineering, contains nine diagrams with dozens of modeling elements. Moreover, profiles like SysML define structure of these systems, but do not contain any information about the model of computation and thus the same SysML diagrams could represent different designs [Lee, 2010]. Therefore, we believe[2], that in our case, the meta-modeling approach of DSM is more desirable.

[2]There is an outgoing discussion about the advantages and disadvantages between the DSM and UML within the modeling community. Interestingly, however, the panel discussion held at MODELS 2012 conference titled *Unified vs. domain-specific: should we have fewer or more modeling languages?* resulted in 80% of the participants voting towards more languages [Tolvanen, 2012].

3.1.3 Why an Actor-Oriented Design?

The use of model-driven software development involves development of a domain-specific modeling language and transformation tools such as a domain-specific code generator and a domain framework[3] [Kelly and Tolvanen, 2008]. The modeling language should map directly to the problem-level abstractions while the generated code has to map to the target environment. The *abstraction barriers*[4] that has to be implemented for the domain-specific modeling are therefore at two different levels [Kelly and Pohjonen, 2009]: (1) at the DSML level, between the models and the generated code, and (2) at the domain framework level, between the generated code and the target libraries and platform. There is a close relationship between these levels and therefore there is always a trade-off between them and in general between the language domain-specificity and versatility. The more specific the models are, the less variability they cover resulting in more commonalities in the system. This requires more effort in building the domain framework, the code generator or both. On the other hand the less specific and more versatile models are harder to design and evolve, but less work is put into the synthesis of the implementation as the modeling concepts map more naturally to the concepts of the target programming language.

Modeling Feedback Control Loops One of the main problems in modeling control systems is how to decompose them into more manageable and domain-specific subsystems to effectively divide and conquer the problem [Liu et al., 2004]. In Sections 2.1.1 we have shown a general feedback control loop decomposition into conceptual processes such as monitoring, decision-making and reconfiguration (*cf.* Section 2.1.2) or MAPE-K (*cf.* Section 2.1.3). Essentially, a feedback can be seen as a sequence of interconnected processes that have inputs, outputs and encapsulate some state and some behavior. For example, the block diagram (*cf.* Figure 2.1) used in control theory represents these processes as system blocks representing functions that transforms the signals floating between them. Such a decomposition naturally maps into many object-oriented software component models. Indeed, many of the framework-based approaches that we have shown in the literature overview (*cf.* Sections 2.2.1 and 2.2.2) are using some sort of components to build self-adaptation, for example [Garlan et al., 2004, Asadollahi et al., 2009, Romero et al., 2010, Chen and Kotz, 2002]. There is a trade-off in using a component model as a base for a DSML. While it simplifies its development by reusing existing component model concepts and implementation for domain frameworks, in turn it might lower the available abstraction level.

Components A software component is generally defined as an element encapsulating a set of related data and operations through some well-defined contractual interface [Szyper-

[3] The domain framework is optional and the code generator can output code without the need of a framework. This is usually done in the case where the code does not require any framework or when the target language has limited structuring facilities. The two different code generation are sometimes referred to as *model-aware* and *model-ignorant* generation [Fowler, 2010]

[4] The very same concept has been introduced in SICP book by Abelson and Sussman [Abelson et al., 1996, Section 2.1.2]

ski, 2002]. It is also the main unit of composition and multiple related components can be composed together in order to form higher-level structures, enabling systematic approach to system construction. In our case, components should represent the fine-grained processes of feedback control. Component models allows to use the divide-and-conquer pattern and split the feedback control processes into a tree. Each conceptual process can therefore be decomposed into smaller units until it becomes small enough to be handled directly in one piece, *i.e.*, inside a component. The resulting process is therefore not only clearly structured, but also easier to reason about and simpler to implement. Such a decomposition should also foster reusability as there is a higher chance for a small component (*e.g.* dedicated sensors) to be usable across different adaptation scenarios. Furthermore, component models provide strong mapping between architectural concepts and implementation concepts which is one of our requirements.

There are different component-based designs such as object-orientation or middleware-orientation [Szyperski, 2002]. The object-oriented design is based on object and class abstractions with interfaces defining how the objects can be interacted with. The middleware-oriented designs emphasize the encapsulation of multiple objects into conceptual services. The notion of a service is especially powerful in the distributed systems since it provides a higher-level abstraction with better non-functional property handling over regular *Remote Procedure Calls* (RPC) in general object-oriented frameworks. However, both designs are based on objects that are related to each other by references and their main interaction is done through method calls that directly transfer the flow of control from one object to another [Liu et al., 2004]. Consequently, some important characteristics of the system behavior such as concurrency and communication remains hidden behind the method interfaces.

Concurrency Feedback is by definition a reactive process (*cf.* Section 2.1.1 on page 12) that activates upon some stimulus sensed in the system. Even a single loop feedback control may employ multiple sensors that react upon different changes in the environment and therefore making the system inherently concurrent. Moreover, as we discussed at the beginning of this chapter, self-adaptive systems might likely involve remote communication which usually also involves concurrency. However, concurrent programming is known to be difficult [Sutter and Larus, 2005]. In most of the mainstream programming languages used for building component-based systems, the usual support for concurrency is based on threads. The main problem with threads is that they are widely nondeterministic, and despite numerous toolkits helping to effectively prune this nondeterminism, concurrent programs remain difficult to develop and to reason about [Lee, 2006]. This is not satisfiable as it will be left to developers to deal with concurrency and we want to simplify the implementation effort.

The Actor Model Another component-based design is the actor-oriented design that provides an alternative to the traditional approach to concurrency using shared memory with locks. The actor model extends the concept of objects to concurrent computation

based on *actors*. It has originated to describe a concept of autonomous reasoning agents by Carl Hewitt in 1970's [Hewitt, 1977] and was further evolved into a formal model for concurrent computation by Agha and others [Agha, 1990]. Like objects, actors also encapsulate their state and behavior. However, while objects primarily interact by transferring control through method calls, actors interact strictly by exchanging messages [Lee, 2003]. Messages are sent asynchronously and each actor maintains a queue of received messages (*a mailbox*) and processes them one-by-one.

Actors act independently of one another. In its core an actor is a simple entity that can only receive, process and send messages. The actor state is implemented using the ``*share nothing*'' policy. It is stored within the actor itself, never being globally accessible. An actor can only manipulate its own state and never directly accesses the state of other actors, however, it can send a message that in turn may change the state of its recipient. In response to a received message it can: (1) send a finite number of messages to other actors, (2) create a finite number of new actors, and (3) designate the behavior to be used for the next message it receives. The actor model is inherently reactive and each actor waits for messages in order to do its work.

Following is a summary of the most relevant features of the actor model in our context:

1. *Thread safety.* One of our aims is to simplify the development of feedback control loops. The actor model guarantee that actor message processing is always executed by one thread at a time. This allows one to implement code without worrying about thread safety[5] which greatly simplifies code [Lee, 2006]. This is a notable advantage over the standard shared memory concurrency. Accessing an actor's mailbox is by design *race condition* free and therefore potentially more secure than shared-memory concurrency with locks. It is important to note, however, that actors are not *deadlock* free (two or more actors are waiting on each other to progress) neither the actor design eliminates *starvation* (one or more actor cannot make a progress because of the inability to access a certain resource) or *livelocks* (like deadlock, but instead of being frozen in a state of waiting for others to progress, the actors continuously change their state).

2. *Scalability.* In general, the actor model has a lower context-switching overhead over the standard shared-memory threads with locks [Haller and Odersky, 2009].

3. *Distribution.* The message-based concurrency with encapsulated state can be seen as a higher-level model for threads. It can be generalized to distributed computation without the necessary cost of creating the illusion of shared memory [Haller and Odersky, 2009].

4. *Dynamic Nature.* Actor based system are dynamic in their nature and they can evolve at runtime. An actor can dynamically create any number of new actors, its behavior may change over time and it is free to send messages to any acquaintances.

[5] Provided that the passing messages are immutable.

5. *Programming support.* Today, there exists several high-performing actor libraries for mainstream programming languages[6]. Constructing our approach on the grounds of an actor model with clearly specified rules about the communication rules and guarantees makes it possible to target multiple implementations in variety of programming environments.

The actor-oriented design has been successfully used in many system-design platforms and development environments [Liu et al., 2004]. For example Mathworks's Simulink[7], National Instruments's LabVIEW[8], Synopsys's System Studio[9] or the already mentioned Ptolemy project (*cf.* Section 2.2.2 on page 29).

3.2 Running Example

Throughout the rest of this thesis we will use the following adaptation scenario to better illustrate our approach. It is based on the work of Abdelzaher *et al.* [Abdelzaher and Bhatti, 1999, Abdelzaher et al., 2002] on QoS management control of web servers by content delivery adaptation. The reason for choosing this particular work is that (1) it provides a control theory-based solution to a well-known and well-scoped problem, and (2) it provides enough details for its re-engineering. The authors also regard more complex issues such as performance isolation and service differentiation. For the simplicity, in this example, we only consider the case with a single server and with all requests having the same priority.

The aim of the adaptation is to maintain the server load at a certain pre-set value preventing both under utilization and overload. The content of the web server is preprocessed and stored in M content trees where each one offers the same content but of a different quality and size (*e.g.* different image quality). For example let us take two tress `/full_content` and `/degraded_content`. At runtime, a given URL request, *e.g.* `photo.jpg`, is served from either `/full_content/photo.jpg` or `/degraded_content/photo.jpg` depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps reducing the server load when the server is under heavy load.

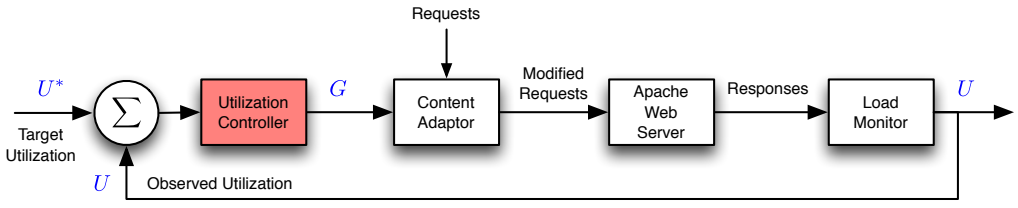


Figure 3.1: Block diagram of the running example [Abdelzaher and Bhatti, 1999]

[6] http://en.wikipedia.org/wiki/Actor_model#Actor_libraries_and_frameworks
[7] http://www.mathworks.com/products/simulink/
[8] http://www.ni.com/labview
[9] http://www.synopsys.com/Systems/BlockDesign/Pages/default.aspx

Figure 3.1 shows the block diagram of the proposed control. The *Load Monitor* is responsible for quantifying server utilization U. It does that by periodically measuring request rate R and delivered bandwidth W. These measurements are then translated into a single value, U. The *Utilization Controller* is a *proportional integral* (PI) controller, which based on the difference between the target utilization U^* (set by a system administrator) and the observed utilization U, computes an abstract parameter G representing the severity of the adaptation action. This value is used by the *Content Adaptor* to choose which content tree should be used for the URL rewriting. The achieved degradation spectrum is shown in Figure 3.2. It ranges from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected.

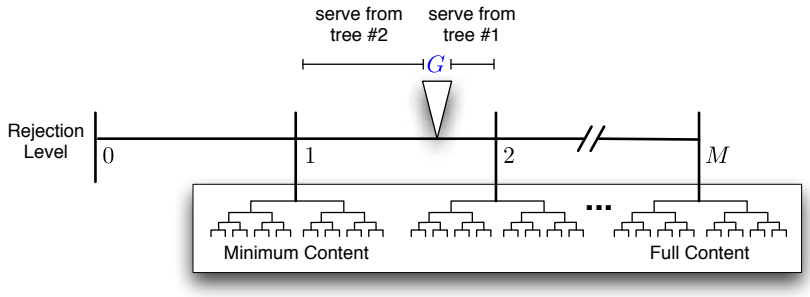


Figure 3.2: Degradation rate for content tree selection [Abdelzaher and Bhatti, 1999]

A service time of a request constitutes of a fixed overhead and a data-size dependent overhead. Therefore, using some algebraic manipulations, the utilization from the request rate and delivered bandwidth is derived as

$$U = aR + bW = a\frac{r}{t} + b\frac{w}{t} \tag{3.1}$$

where a and b are some platform constants derived by server profiling, r is the number of request and w is the amount of bytes sent over some period of time t. The authors propose a simple way to profile a web server by subjecting it to an increase request rate and estimate its maximum processing capacity, for which the error rate is less than 0.1%. With different URL sizes, each profiling yields a pair of R_{max}, W_{max} being respectively the maximum achievable request rate and the corresponding bandwidth. These numbers represents the state of a fully utilized server $U = 100\%$ and therefore using linear regression the a, b parameters can be derived as $100 = aR_{max} + bW_{max}$.

The abstract parameter G is computed as

$$G = G + kE = G + k(U^* - U) \tag{3.2}$$

Shall $G < 0$ then $G = 0$ and similarly shall $G > M$ then $G = M$. If the server is overloaded $(U > U^*)$ than the negative error will result in decrease of G which in turn might change the content tree decreasing the server utilization and vice versa. The k is a tuning parameter of the PI controller that has to be determine *a priori* usually by using some of the analytic techniques for the best convergence [Kaczorek, 1993].

While the presented solution is technologically agnostic, in the rest of the thesis we will consider Apache Web Server[10], which is today the most deployed web server [Netcraft, 2013].

3.3 Feedback Control Definition Language

In this thesis we propose a domain-specific modeling language for developing self-adaptive software systems called *Feedback Control Definition Language* (FCDL). The purpose of this language is to allow researchers and engineers to expressively and concisely define amenable feedback control architectures for external self-adaptive software systems using feedback control loops. The architecture is defined as a set of higher-level structural elements representing the feedback processes and the relations between them. In this section we present the FCDL meta-model that defines the abstract syntax of our modeling language. The abstract syntax will be accompanied with the semantics definitions in the next chapter.

We start with a high-level overview of our DSML approach using the running example introduced in the previous section. The rest of the subsections detail the FCDL concepts and its meta-model.

3.3.1 High-Level Overview

FCDL is a reactive actor-oriented component model representing abstractions of feedback control loop architectures. The components in FCDL are actor-like[11] entities called *Adaptive Elements* (AE). An architecture is created by assembling and connecting the adaptive elements into hierarchically composed networks that form closed loop feedback control system.

Adaptive Element Like actors, adaptive elements have a well-defined interface that abstracts their internal state and behavior and restricts how they interact with their environments. The interface defines input and output ports that are the points of communications. Connecting an output port to some input port creates a communication link through which elements can exchange messages. Once an adaptive element receives a message, it activates and executes its associated behavior. The execution can be regarded as function $input \times state \rightarrow output$. The result of the function computation may or may not be sent further to the connected downstream elements that would in turn cause them to active and so forth. Next to ports, adaptive elements can also define properties that are configuration options of the element's operation.

In general an adaptive element can define any number of ports and the model does not place any restriction on its behavior. However, each adaptive element represents a process of a feedback control loop, which may either be: *a sensor*, *a processor*, *a controller* or *a effector*. Sensors are the data sources in the system having only output ports. Analogically, effectors are the data sinks having only input ports. They both represent the target system

[10] http://httpd.apache.org/

[11] The are domain-specific actors providing a higher-level interface than the one of base actors.

touchpoints and they are effectively the only points of interaction with the underlying target system. Processors and controllers on the other hand have both inputs and outputs processing data along the data flow path from sensors to effectors. Similarly to other component models, FCDL also allows to construct composite components from both basic adaptive elements and from other composite components. A composite in FCDL is the primary unit of deployment. They define both the instances of other components they contain and the connection between the instances ports (*cf.* Section 3.3.4).

Feedback Control Loop A feedback control loop is realized by connecting adaptive elements ports together into a network. According to elements roles, this network can be partitioned into two layers:

– the *system layer* that consists of sensors and effectors, *i.e.* the target systems touchpoints, providing all the necessary inputs and outputs for the adjacent

– *control layer* that uses the sensors data (inputs) to infer the state of the target system, reasons about it and computes the required adjustments that will be put into effect by the system layer effectors (outputs).

The sensors and effectors from the system layer are realized using management interfaces provided by the target system. Essentially, they shall be implemented as thin wrappers over these interfaces, operating at the same level of abstraction. For example, they can wrap a system command (*e.g.* `kill` command terminating a system process), a JMX MBean (*e.g.* the `MemoryMXBean`[12] for getting information about the memory system of the Java virtual machine) or a log file. While it might seem to be too low-level [Vogel and Giese, 2010], there are two reasons for this choice. First, they should not try to reduce or hide complexity of the underlying system, by building any higher abstraction over the one that is provided by the operational interface. It is the responsibility of the processors from the control layer to infer higher-level information from the low-level sensor data and analogically for the reconfiguration part. Second, being thin wrappers simplifies reasoning about their functionality and increases their chances of being reused.

Next, we illustrate our approach by implementing the complete running example (*cf.* Section 3.2) in FCDL.

Illustration Figure 3.3 shows the FCDL model of the running example. The architecture is derived from the block diagram presented in Figure 3.1. Circles represent adaptive elements with the inner glyph indicating the element role. Arrows denote the communication links over which data are transported using either *push* or *pull* style (indicated by the direction of the arrow).

– *The decision-making part* of the control loop represents the adaptation engine. It has been elaborated in Section 3.2 and our aim is to integrated it into the self-adaptive software system that we implement. Based on a PI controller (*Utilization Controller*

[12] `http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/management/MemoryMXBean.html`

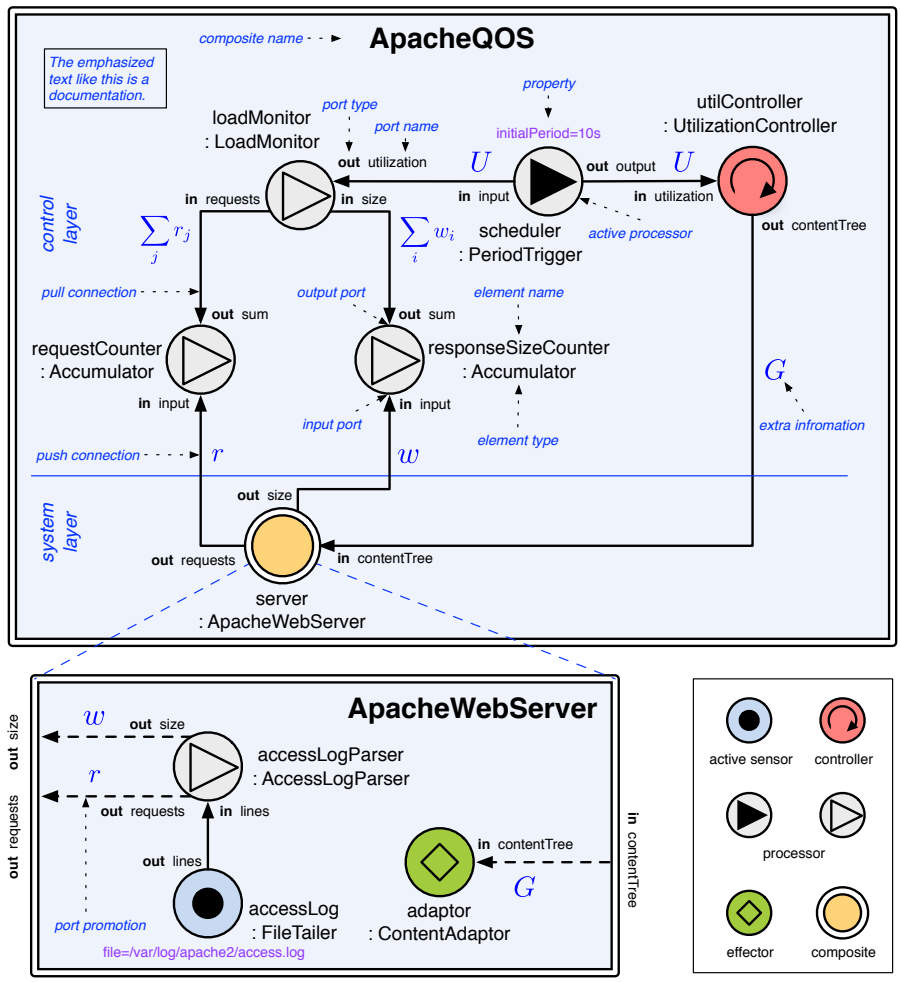


Figure 3.3: FCDL schema of the running example

from the block diagram) it maps the current system utilization characteristics U into the abstract parameter G controlling which content tree should be used by the web server. In FCDL it is represented by the `UtilizationController` controller that has one push input port, `utilization`, for U and one push output port, `contentTree`, for G. Once a new utilization value is pushed to its input port, it computes G using equation (3.2) and pushes the result to the output port.

– *The monitoring part* is therefore responsible for providing the controller with the system utilization metric U. The utilization depends on request rate and bandwidth (3.1). In the case of Apache web server, both information pieces can be obtained from its access log. By default Apache logs all requests it processes into a text file with one line per processed request including the size of response in bytes[13]. To get this information from the log file, we need to first create an active sensor, `FileTailer`, that activates ev-

[13] `http://httpd.apache.org/docs/2.0/mod/mod_log_config.html#logformat`

ery time a content of a file changes[14] and sends the new lines over its push output port. This output port is connected to `AccessLogParser` that once activated parses the incoming lines and computes the number of requests r_j and the size of the corresponding responses w_i. It pushes these two values to the appropriate `requests` and `size` ports. Consequently this increments the values of two connected counters `requestCounter` and `responseSizeCounter`. These accumulators are simple passive processors. When they receive data on their `input` port, they simply update their state, *i.e.*, add the received value to the sum of all values received so far.

At this point we have the accumulated information about the total number of requests $\sum_j r_j$ and the size of responses sent out $\sum_i w_i$ from the time the feedback control system has been launched. In order to compute U, we need to convert these values to request rate R and bandwidth W, *i.e.*, the number of request and sent bytes over certain period of time t. One way of doing this is by adding a scheduler that would periodically trigger the computation of U. We therefore create a new active processor, `PeriodicTrigger`, that every t milliseconds (initially set by the `initialPeriod` property) pulls data from its pull input port and in turn pushes the received data to its output port. Essentially, it acts as a mediator between two connected adaptive elements controlling the timing of data requests and distribution. In this scenario, it is responsible for the timing of feedback control loop execution. By pulling data from its input port, it will activate the `LoadMonitor` processor that will in turn do the following: (1) fetch the corresponding sums of requests $\sum_j r_j$ and response sizes $\sum_i w_i$ using the two pull input ports; (2) convert them to requests rate R and bandwidth W; and (3) finally compute the current system utilization U using equation (3.1). The resulting utilization is then forwarded by the scheduler into the `UtilizationController`.

– *The reconfiguration part* of the loop is represented by the `ContentAdaptor`. When it receives the extent of adaptation G into its push input port `contentTree`, it reconfigures the web server URL rewrite rules so to use the newly computed content tree. One way of implementing the dynamic URL rewriting is to use a *Common Gateway Interface* (CGI) script to handle all the traffic. The content tree reconfiguration is then simply a matter of notifying the script that the content tree has changed (*e.g.* using a text file).

On the Implementation Variability It is important to note that there exists other ways to model the architecture of the running example and to implement the interaction with the Apache web server. The above model and implementation is just one possibility. It has been chosen because (1) it is simple, (2) close to the original implementation [see Abdelzaher and Bhatti, 1999, sec. 4] and (3) convenient for illustrating some of the FCDL features in the coming section. For example, the `LoadMonitor` could also be an active processor with an internal scheduler, but decomposing the functionality into two elements makes a clear separation of concerns and increase reuse. The `PeriodicTrigger` provides a generic scheduling facility that can be used across different adaptation scenarios based

[14] Similar to the unix ``tail -f'' functionality http://en.wikipedia.org/wiki/Tail_(Unix)#File_monitoring.

on periodic observation. Similarly, by splitting the access log file observation and parsing into two adaptive elements, we make the `FileTailer` a general component that can also be reused across multiple adaptation use-cases.

Later, in Section 4.1.4, we will show why the proposed solution might be the preferred one. Nevertheless, one of the aim of the architecture model is to allow to experiment with different designs leveraging from the fine-grained reusable adaptive elements.

To demonstrate composition, the presented elements are assembled into two composites `ApacheQOS` and `ApacheWebServer`, representing respectively the control part and the target system touchpoints. Throughout the following text this assembly will be further refined into a more fine-grained and simpler organization.

Graphical Notation and Meta-Model Presentation The concepts of adaptive elements, ports, properties and communication links describe the abstract syntax of the model. An abstract syntax can have multiple concrete representation. In this chapter we use an informal graphical notation to visualize the feedback processes and the data flowing among them, much like block diagrams (*cf.* Section 2.1.1). The complete notation is shown in Appendix B. Its purpose is to provide an intuitive and expressive visual representation of the model that can be easily sketched by hand. A formal FCDL textual representation is given in Section 5.1.2.

In the following text we present excerpts of the FCDL meta-model as UML class and object diagrams. The full meta-model class digram is included in the Appendix B.2 and B.3. A gray color is used for the elements that have been already presented before or will be detailed in a later section. Different colors are also used to make clear distinction between meta-model classes and their instances.

The rest of this section gives description of the FCDL data type system, details about adaptive elements and their composition, reflection and distribution, relation between types and instances, and annotations.

3.3.2 Data Types

The data type system is used to classify the data values within feedback control loops. To make adaptive elements work together, the connected ports must be compatible in some way. In this section we are interested in data type compatibility.

To enforce data type compatibility, the FCDL modeling language is using static typing. It allows type safety verification at design time and therefore it prevents typing errors [Cardelli, 1997]. There are two FCDL entities that are concerned with data types: ports and properties. For each port and property one has to explicitly declare the data type that restricts the data values it accepts. Based on these information, the model is checked for data type conformance.

Data Type System The FCDL modeling language aims at being technologically agnostic model. The data type system therefore has to be generic enough to be translatable to type

systems of the various programming languages. One way of doing this is to define a complete type system that consists of a number of primitive types (*e.g.*, floats, integers, strings) and constructed types (*e.g.* arrays, enumerations, structures) like Ptolemy [Xiong and Lee, 2000] or CORBA [Object Management Group, 2012a]. The main drawback is that it is usually difficult and makes the language bigger and more complex.

Instead of defining a complete type system, we have chosen to rely on a target programming language for the type conformance verification. The data types of ports and properties are specified using arbitrary names and for each name a developer provides a mapping to a concrete data type for a given programming language. Then, for each target programming language we have to provide a data type verifier that checks whether two types are compatible by relying of the facilities provided by the target language (*e.g.* `Class.isAssignableFrom` in case of Java). A concrete mechanism is presented in Section 5.1.

While developing a type conformance verifier based on a programming language infrastructure is arguably easier than developing a complete type system, there are some shortcoming to this approach as well. For each target programming language we have to develop a new verifier. The effort greatly depends on what facility is being provided by the actual language and developers have to specify the extra mappings. Further, since there are no predefined data type names an inconsistency across models can occur (*e.g.* in some models a developer might choose `int` for integers in others `int32`).

Based on our experience so far, we believe that the advantages of this approach, namely the simplicity, over the custom data type system development outweight these shortcomings.

Data Type Representation Figure 3.4 shows the data type system representation in the FCDL meta-model.

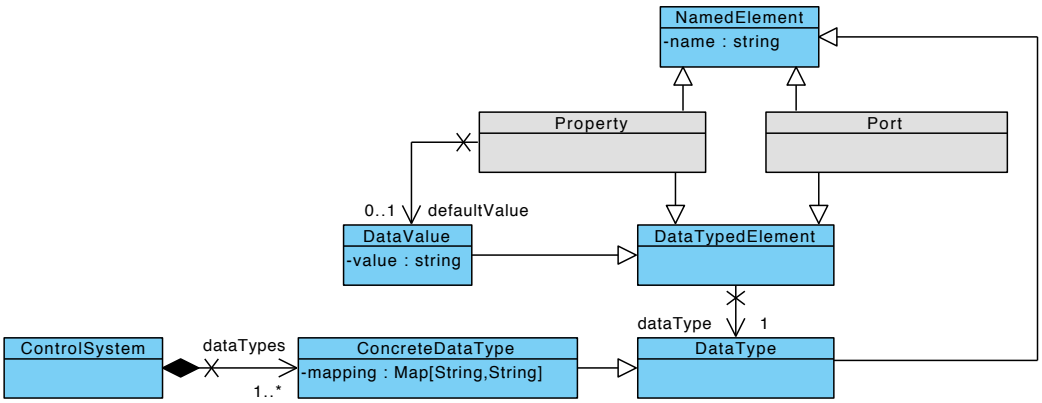


Figure 3.4: FCDL meta-model excerpt related to data type system

`ControlSystem` is the top level element within the FCDL meta-model `types` package representing a control system. It groups together concrete data types and related adaptive elements types (*cf.* Section 3.3.3).

`NamedElement` is used as a super class for any model elements that are identified by name.

`DataType` represents a FCDL data type.

`ConcreteDataType` models a concrete data type with appropriate mapping to different programming languages.

`DataTypedElement` is a super class inherited by any model element defining a data types, *i.e.*, a port and a property.

`DataValue` represents a data value encoded as a string. The serialization mechanism is external to the model since the model does not use these values in any way.

Polymorphic Adaptive Elements To improve reusability, the meta-model also supports parametric polymorphism [Cardelli and Wegner, 1985], making adaptive elements work uniformly on a range of data types. For example, the `PeriodicTrigger` element from the running scenario should be able to pull/push any data. This is realized by using a data type parameter for the data type definition of its ports. Essentially, a data type parameter is a placeholder for a concrete data type that will be specified at some point later. A data type parameter can be used instead of concrete data types anywhere a data type definition is required. The actual data type is then specified when the element is declared in a composite using a data type argument. Depending on the nesting level it can either refer to concrete data type or to another data type parameter. The top level composites cannot define any data type parameters and therefore at the latest all parameters are resolved at this level.

Unlike type variables like Java generics or Scala parameterized classes, the data type parameters in FCDL do not provide any further assumption about the type itself (*e.g.* operation it provides). There is no support for any type constraints in FCDL. We leave this to the compiler that will raise a compile time exception on an attempt to compile code generated from an incorrectly assigned data type arguments.

Figure 3.5 shows how the data type parameter is represented in the meta-model. It will be further discussed in the following section when we present details about adaptive elements and their composition, *e.g.* Figure 3.10 shows a concrete example of a `PeriodicTrigger` object model.

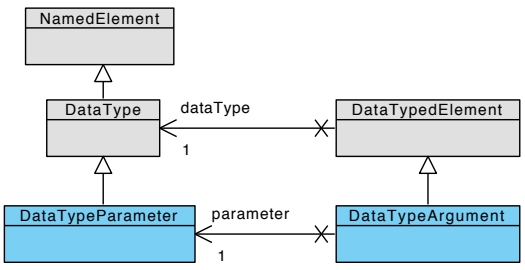


Figure 3.5: FCDL meta-model excerpt related to polymorphic data type system

3.3.3 Adaptive Element

The FCDL meta-model consists of two packages: `types` and `instances`. The former is used to define adaptive element types, *i.e.*, the elements structure such as ports and properties, while the latter defines adaptive element instances, *i.e.*, configured adaptive elements. In this section we focus on FCDL types. Instances are covered later in Section 3.3.7.

An adaptive element type is the basic component within FCDL. It is an actor-like independent entity that encapsulates its state and behavior, and communicates with other adaptive elements only via message passing through its ports. Figure 3.6 presents an excerpt of the `types` package related to the adaptive element type definition. It consists of the following elements:

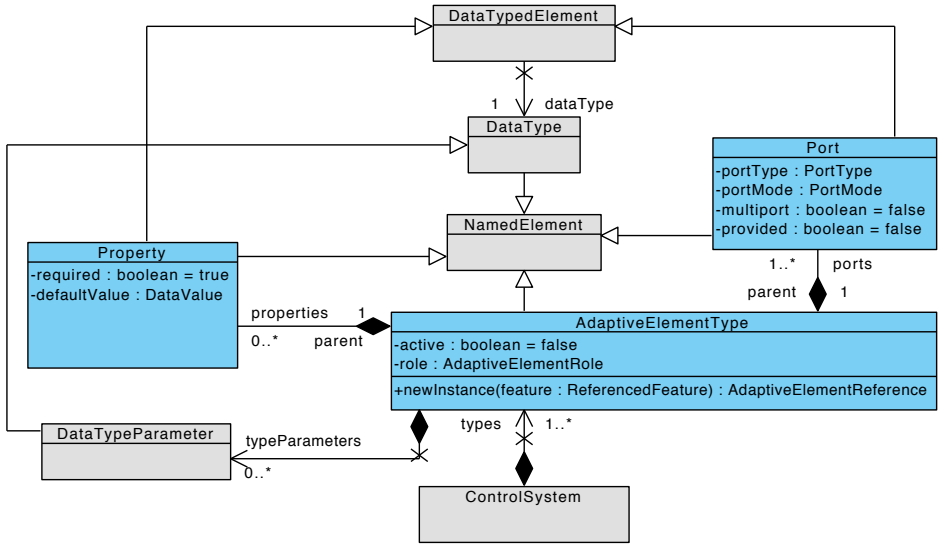


Figure 3.6: FCDL meta-model excerpt related to adaptive element types

`AdaptiveElementType` models an adaptive element type. Next to the inherited *name* attribute, it has a *role*, can be either *passive* or *active* and have one or more *ports*, multiple *properties* and *data type parameters*. The role attribute defines the intent of the element. Following roles are defined in FCDL:

- *Sensors* are elements with no inputs that collect raw information about the state of the running system and about its environment (*e.g.*, using operating system probes, services calls, log files).
- *Effectors* are elements with no outputs that carry out changes on the running system using provided management operations (*e.g.*, operating system commands, service calls, configuration file modifications).
- *Processors* are elements with both inputs and outputs. They are used to process and analyze incoming data (*e.g.*, filter, stabilize, convert) or to drive the data flow and manage storage properties (*cf.* Section 4.1). They can be used both for the

monitoring part as well as for the reconfiguration part. For example a data stabilization filter can be used both to stabilize a sensor output or an effector input.

– *Controllers* are special cases of passive processors that are directly responsible for the decision making process. Essentially, they are choosing the appropriate actions based on the current target system state that has been inferred by monitoring part in order to get it into some desired state.

The remaining *composite* role is discussed in the next Section 3.3.4.

`Port` Adaptive elements communicate with one another via messages that are sent and received through ports. They represent the adaptive elements inputs and outputs (according to the *port type* attribute). The *port mode* denotes the communication style that can be either: *push*, *pull*, or *agnostic* in which case the exact mode is resolved during element instantiation according to the connected ports (*cf.* Section 4.1.6). The FCDL ports are data typed. The *data type* reference specifies what values a port accepts. The *provided* attribute separates regular input and output port from the provided ports used for reflection (*cf.* Section 3.3.5). Finally, the *multiport* attribute denotes whether a port can be connected to more than one peer.

`Property` enables external configuration of an adaptive element behavior without the need of changing its code. It has a *name*, a *data type*, a *default value* and it can be either *required* or *optional*. The value of a property is specified when the adaptive element is declared in a composite (*cf.* Section 3.3.4) and all required properties without default values must be specified. Properties can be though of as constants. They are usually used as initial values for adaptive element configuration which can be further altered at runtime using reflection (*cf.* Section 3.3.5).

Active Adaptive Elements An adaptive element can be either passive or active. A passive element executes upon a demand initiated by receiving a message on any of its ports. An active element can additionally activate itself explicitly in a response to an external event (*e.g.*, a file changed, a timeout). It is associated with an event handler that is notified when such an event occurs. When this happens, the event handler activates the associated element by sending it a message though a special port called `selfport` which is essentially a push input port implicitly defined for each active adaptive element. The data sent though this port are usually the context information associated with the event. The reason for this separation is that an adaptive element is an actor and as such it can only be activated by receiving message. Any change to this model would break the encapsulation and thread safety guarantees.

Listing 3.1 shows an excerpt of the `PeriodicTrigger` implementation in Scala to better illustrate the self-activation.

```
1 // instance of the implicit self port
2 val selfport: SelfPort[Long]
3
4 var trigger: Cancellable = _
5
```

```
/** Adaptive element initialization */
def init {
  trigger = context.scheduler schedule (delay = 2 seconds, period = 5 seconds) {
    // event handler code that will be called by the scheduler thread
    // the data sent are the context information associated with the event
    // in this case it is simply the current time
    selfport put System.currentTimeMillis
  }
}

/** Adaptive element activation */
def activate {
  // the activation method is called by the actor framework when the element
  // receives a message on any of its ports

  // the value pushed over the self port is exposed via selfport variable
  logger debug s"Activated at: {selfport.get}"
}

/** Adaptive element destruction */
def destory {
  trigger.cancel
}
```

Listing 3.1: Example of adaptive element self-activation

Figure 3.7 shows an example of an adaptive element type definition, concretely the `PeriodicTrigger` from the running example. It is an active processor defining three ports (`input`, `output` and `selfport`) and one property (`initalPeriod`).

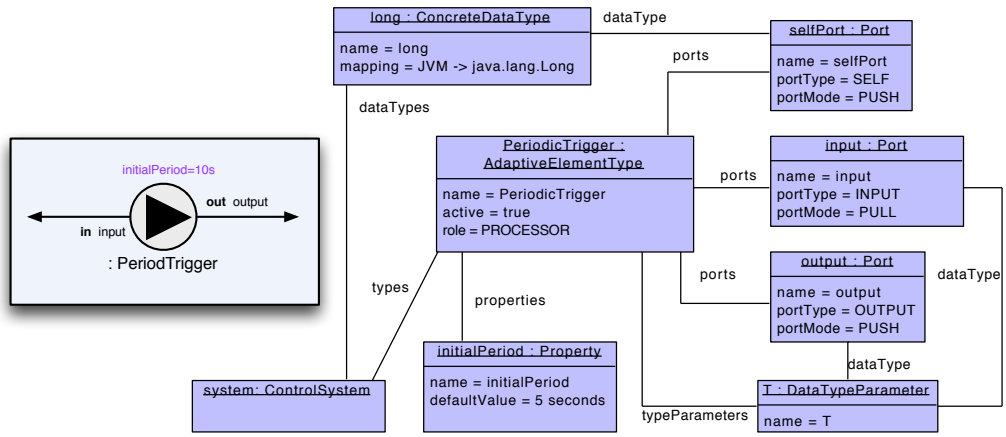


Figure 3.7: Example of the of `PeriodicTrigger` adaptive element type definition. The left part in the box is the FCDL graphical notation, the right part shows instance of the element from the FCDL meta-model `types` package defining the element.

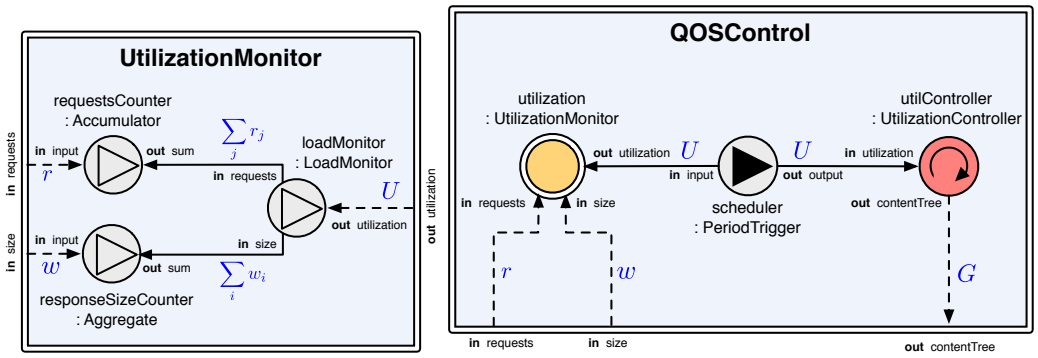
3.3.4 Composition

FCDL supports hierarchical organization of adaptive elements using composites. A composite is an adaptive element created as an assembly of other adaptive elements including other composites. For example, the architecture of the running example (*cf.* Figure 3.3)

contains two composites: `ApacheWebServer`, which represents the running web server, providing all the necessary inputs and outputs for the connected `ApacheQOS` composite representing the control.

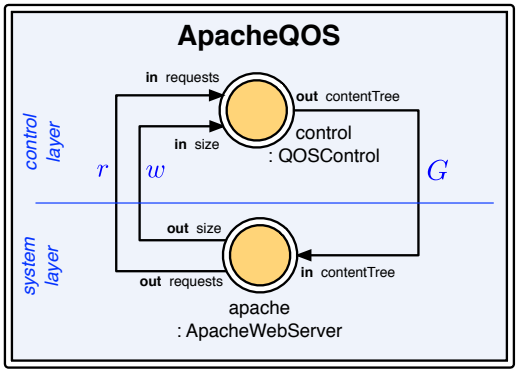
Composites do not define behavior on their own, instead, they define instances of adaptive elements they contain and connection between the ports of the contained elements. The port connections that cannot be satisfied within the composite can be made available to the outside by promoting them to the ports defined by the composite itself.

With the ability to compose adaptive elements, we hope to foster reuse and increase productivity. For example, we can rearrange the model of the running example from Figure 3.3 into a new composition as shown in Figure 3.8. The main advantage of the new organization is that it completely separates the utilization monitoring (`UtilizationMonitor`), the control (`QOSControl`) and the target system (`ApacheWebServer`). We can now reuse the control composite with different web servers just by swapping the `ApacheWebServer` composite for a different one, *e.g.*, `LighttpdWebServer`[15]. Similarly, we could the switch monitoring part responsible for computing the utilization characteristics U to test different strategies. This new arrangement will be used as the main implementation for further illustrations.



(a) `UtilizationMonitor` composite

(b) `QOSControl` composite



(c) `ApacheQOS` composite

Figure 3.8: FCDL model of the running example with composites

[15] `http://www.lighttpd.net/`

Figure 3.9 shows the FCDL meta-model related to composition, containing the following elements:

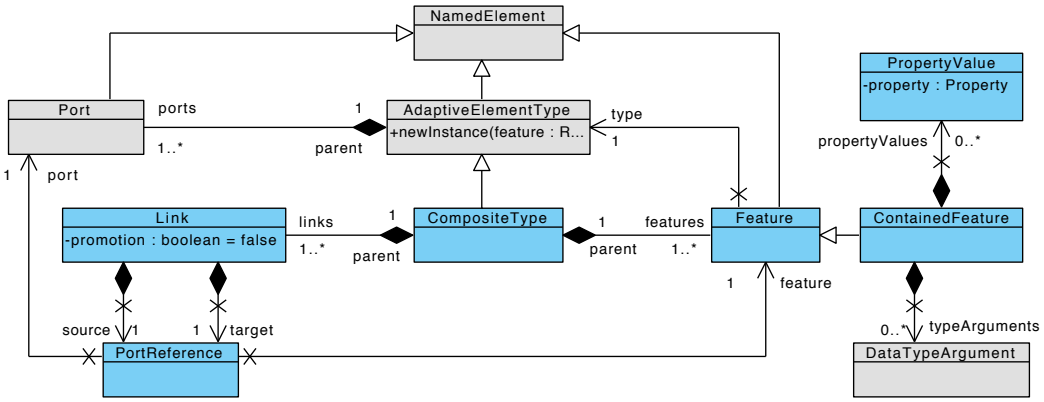


Figure 3.9: FCDL meta-model excerpt related to adaptive element composition

`CompositeType` models a composite. It is an adaptive element and as such it has a name, a role that is always *composite* and it can define a number of ports, properties and data type parameters. Additionally, it defines features and links.

`Feature` represents a declaration of an adaptive element type, denoted by the `type` reference.

`ContainedFeature` defines a configured feature of an adaptive element type that is contained[16] within the composite type. The configuration includes specification of all required properties and data type arguments for all data type parameters that the adaptive element type declares.

`PropertyValue` associates a property declared in an adaptive element type to a data value.

`PortReference` is a pair of a feature and a port that is defined by the feature adaptive element type.

`Link` connects two ports establishing a communication link for messages. It is specified as a pair of port references, a source and a target. There are two types of links denoted by the *promotion* attribute: (1) *connection*, which is a link between two ports defined by the contained adaptive elements, and (2) *promotion*, which is a link between a port of a contained adaptive element and a port defined in the composite itself. A *connection* between a source and a target port is possible if the source is an output port, the target is an input port, the source port data type is assignable to the target port data type, the port modes are the same, neither the source nor the target port is promoted, and if any of the ports is not a multiport then there must be no other connection involving this port. A *promotion* is valid if the port type, port mode, data type, provided and multiport attributes are the same for both ports: the source port

[16] A composite can also reference adaptive elements (*cf.* Section 3.3.6)

that is the port to be promoted and the target port declared in the composite. Finally, a port can be promoted only once.

Figure 3.10 shows an excerpt of the FCDL object diagram defining one connection between the `PeriodicTrigger` output port and the `UtilizationController` input port within the `QOSControl` composite.

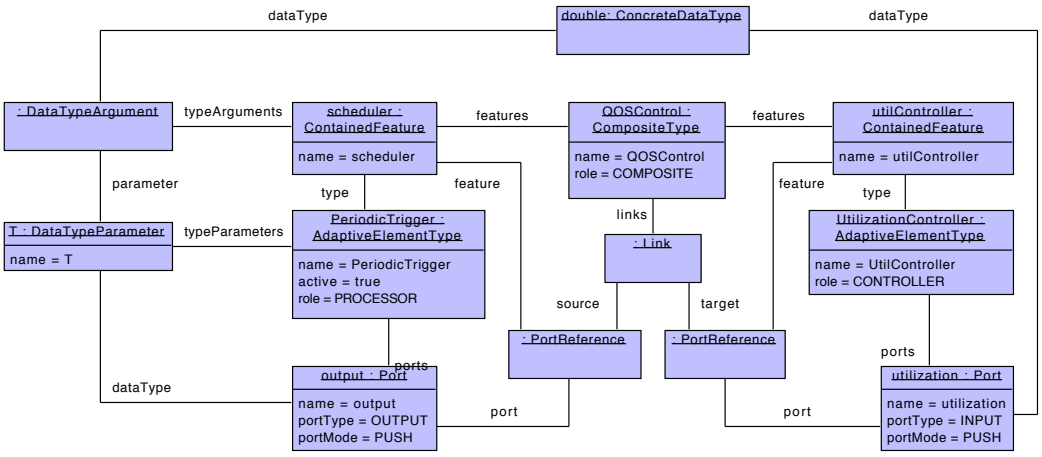


Figure 3.10: An excerpt of the `QOSControl` composite object model

Deployment From the runtime point of view composites are units of deployment. In order for a composite to be deployable, it must not define any data type parameters nor any ports, but it can define properties. These composites are called *main composites*. For example, the `ApacheQOS` is the main and deployable composite from the running scenario. The `ApacheWebServer` composite is not deployable since it promotes ports.

3.3.5 Reflection

In Section 2.1.2 we have discussed the hierarchical organization of multiple feedback control loops. A scheme, where the loops at higher levels influence the loops at lower levels, usually operating at different time scales in order to avoid any unexpected interference. For example: changing dynamically estimates of the model parameters (*adaptive control*), performing on-line optimization of the control model (*model predictive control*), or changing the controller behavior depending on the prior knowledge about the performance variables, disturbances and conditions (*gain scheduling*) [Patikirikorala et al., 2012]. In more complex scenarios, lower level loops manage sub-systems of a large system while high-level loops act as coordinators of the lower level control systems, adjusting their control objectives in accordance with system-wide goals.

In general, the hierarchical control is realized by partitioning the control loops into layers. To realize that, lower layers must provide some sort of reflection at runtime so that loops from higher layers can modify the *parameters* or the *structure* of lower-layer processes. From the adaptation perspective we can therefore recognize two categories *parametric adaptation* and *structural adaptation*.

Parametric Adaptation Conceptually, we can see an adaptive element as a target system itself and as such it can provide sensors and effectors (*cf.* Section 2.1.3 on page 17). This enables adaptive elements to be introspected and modified in the very same way as the underlying running system from the lowest layer. The *provided sensors* and *provided effectors* are essentially the adaptive element touchpoints that make the elements reflective and thereby enabling them to be adaptable[17]. This is a crucial feature that permits one to hierarchically organize multiple feedback control loop in an uniform way and therefore to realize complex control schemes.

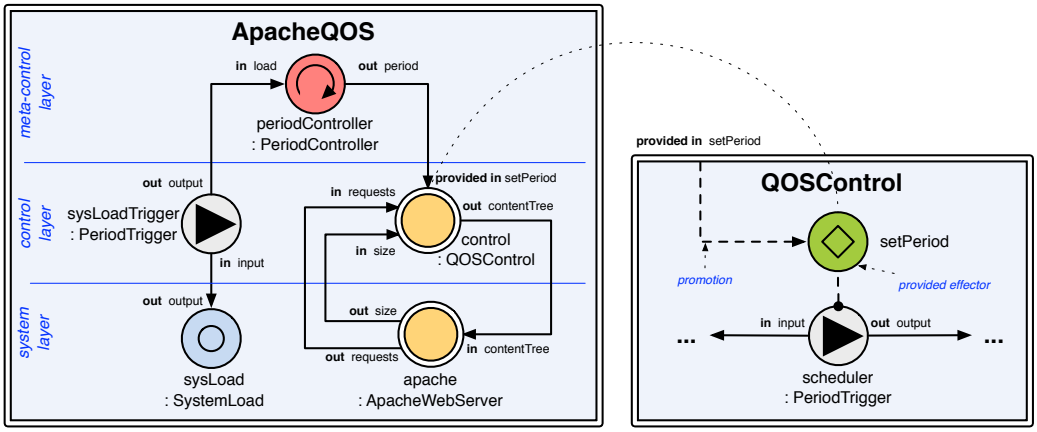


Figure 3.11: FCDL schema of the running example with adaptive monitoring

Figure 3.11 shows an example of an adaptive monitoring added into the running adaptation scenario. Based on a periodically observed current system load using the `SystemLoad` sensor, the `PeriodController` modifies the execution timing of the `QOSControl` using the `setPeriod` effector. The `setPeriod` is a provided effector that adjusts the trigger rate of the `PeriodicTrigger` inside the `QOSControl` composite.

Technically, the provided touchpoints are realized as adaptive elements ports with the *provided* attribute set to **true**. A provided sensors are push output multiports and provided effectors are push input multiports. There is, however, a crucial difference between a regular port and a provided port. The messages sent from or to provided ports have a higher priority and therefore will be processed before the messages transmitted over regular ports. The reason is to put higher significance to the reflection capabilities over the regular element behavior. This is also reflected by their representation in the graphical notation. Provided sensors are visualized as active sensors having one push output multiport and provided effectors are passive effectors with one push input multiport. Both have an additional dotted line indicating to what element do they belong.

Structural Adaptation The adaptive element reflection is suitable for parameter adaptation. A structural adaptation, *i.e.*, changing loop composition and bindings requires

[17] This should also explain why they are called adaptive elements

the underlying actor model to be reflective. This is realized by sensors and effectors that operate on the actor model itself. These touchpoints include sensors observing adaptive elements life-cycles (*e.g.* notifying when a new adaptive element is deployed), effectors deploying new elements or removing the existing ones and changing connections between them. By realizing the model reflection this way, we do not need any particular language support since these touchpoints are just regular sensors and effectors implemented using the underlying framework API.

For example, there are two possible structural adaptation scenarios related to our running example: (1) when the deployment of the `QOSControl` composite is dynamically controlled based on the state of the Apache web server or (2) when there could be more than one web server running at the same time, each having its own control loop. Figure 3.12

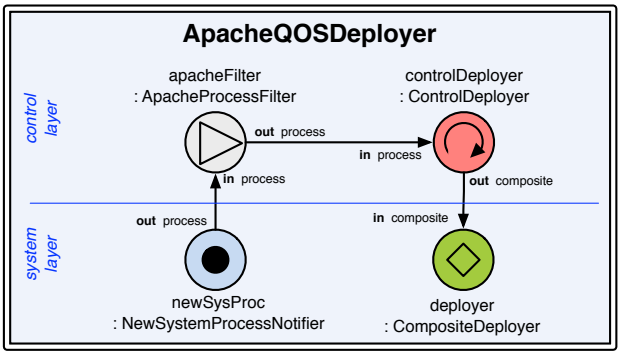


Figure 3.12: FCDL schema of a dynamic deployment of the running example

shows its realization in FCDL. It involves an active sensor that broadcasts notification every time a new system process has been started. The connected processor selects processes that are Apache web servers and pushes them to the controller. The controller uses the `deployer` to spawn a new `QOSControl` composite configured with the correct Apache process (*e.g.* setting the right path to the access log file). Obviously, a similar loop needs to be built for stopping the control loop whose underlying web server has finished execution.

3.3.6 Distribution

It might not be possible to deploy the feedback control together at the same host where the target system is running or the control is managing multiple distributed systems. Therefore, some self-adaptive scenarios might require having parts of the managing system running in different possibly remote runtime environments. For example, in our self-adaptive scenario we need to deploy the control loop outside of the web server and access the web server touchpoints remotely. Therefore distribution is important aspect that has to be addressed in engineering self-adaptive system.

Transparent Remoting An often followed approach to distributed computing is to use *Remote Procedure Call* (RPC). Essentially, it is a inter-process communication that involves executing a program subroutine in a different runtime of the one of the program making

this call often on a different machine. There exist many implementations of this approach in various programming languages[18]. Most of the implementations provide some sort of proxies that when called take care of the input data serialization and its transmission over the network to remote counterparts. They are responsible for calling the appropriate methods, obtaining the results and sending them back to the originators.

In general, many of the RPC implementations intent to provide transparent remoting, allowing to treat remote procedures or objects in the same way as the local ones. In other words, they try to hide the complexity of remote communication over a computer network from the developer allowing to go from local to remote calls with the justification that whether a call is made locally or remotely has no impact on the correctness of the final result. In practice however, because of call latency, different models of memory access and issues of concurrency and partial failure, this assumption does not hold and as a result RPC is a *leaky abstraction*[19]. This issue is discussed in detail by Waldo *et al.* [Waldo et al., 1994].

Location Transparency A different approach to remote communication is by *location transparency* that has been successfully used in Erlang [Armstrong et al., 1992] and other actor based systems such as Akka [20]. It is based on a concept of using ``*logical*'' names to identify network resources and considering all communication between these resources to be remote by default regardless whether they are local or remote. This means that sending a message to an actor that resides in the same runtime is no different from sending it to an actor that is running in a different runtime on a different host. For this to work, all interactions must be purely message based and asynchronous. This also means that there are less guarantees on message delivery. Most of the time implementations only guarantee *at-most-once* delivery, *i.e.*, no guaranteed delivery since a message might get lost due to the inherently unreliable network communications. The key of this approach is to go from remote to local by way of optimization instead of trying to go from local to remote by way of generalization [Typesafe, 2013].

Message Ordering Earlier we have stated that one of the advantage of using an actor system is the implicit thread safety (*cf.* Section 3.1.3). While this also holds in a distributed setting, there are new concerns that have to be taken into an account. For example an actor might assume a certain ordering of messages that can be violated by external non-deterministic effects such as the inevitable network delays that in turn can cause *race conditions*. The way to prevent this is to attach an information about sending order to each message so that the receiving participant can determine that messages have been sent in a

[18] The Wikipedia page about RPC provide a good overview: `http://en.wikipedia.org/wiki/Remote_procedure_call`.

[19] A term popularized by Joel Spolsky in 2002 describing an abstraction that intends to reduce (or hide) complexity while not completely hiding the underlying details. `http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

[20] Akka (`http://akka.io`) is the chosen underlying framework for our implementation, which will be further discussed in Section 5.2.2.

different order. Some actor implementations (*e.g.* Akka) do that by default and thus they guarantee *message ordering per sender–receiver pair*.

Adaptive Element References Being based on an actor model, FCDL supports remoting using location transparency. Remote elements are represented as first class entities using references. Figure 3.13 shows the meta-model excerpt related to element distribution.

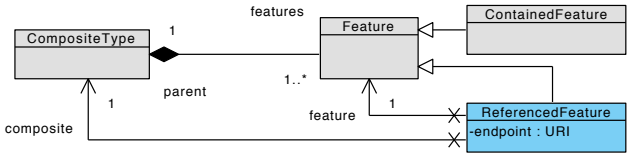


Figure 3.13: FCDL meta-model excerpt related to element distribution

At the composite level, instead of declaring a new contained feature as we have shown in the Section 3.3.4, one can declare a referenced feature (`ReferencedFeature`). A referenced feature is formed by a reference to an existing feature in some composite, and a destination *endpoint* which is a URI [21] of the remotely running adaptive element. At runtime, when the composite is instantiated, for each feature reference it skips creating new adaptive element and instead it only creates an adaptive element reference that points to the given endpoint.

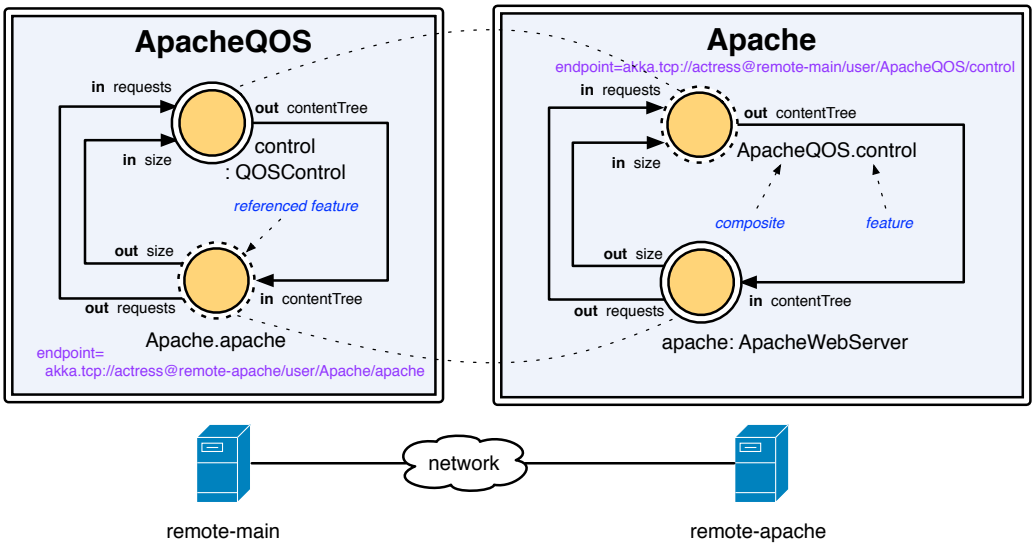


Figure 3.14: Running example with `ApacheWebServer` running remotely

Figure 3.14 shows the referenced feature representation in the graphical syntax. A concrete example of element distribution is given in the second self-adaptive case study that is discussed in Section 8.1.3.

[21] In this example we use the Akka URIs `http://doc.akka.io/docs/akka/2.2.0/scala/remoting.html`

As we have discussed in Section 3.1.2 there are always design and implementation trade-offs between the domain model and the domain framework. In this particular case we can see, that while enabling the remote communication in the model is rather simple, doing the same in the framework can be very difficult. In our case, however, the runtime implementation is built on top of the Akka framework that well supports the principles of location transparency (*cf.* Section 5.2.2).

3.3.7 Instances

The FCDL type meta-model that we have focused on so far provides an expressive and a concise way for developers to define feedback control loop architectures. However, type models are not very convenient for an automated analysis that usually involves traversing instances rather than just types. For this purpose, FCDL defines instance meta-model in the `instances` package.

It is important to realize that both the type and the instance meta-models are defined at the same meta-modeling level. Therefore an instance of an `AdaptiveElementType` is not an adaptive element instance, but it is just a type instance defining a new type (either a basic type such as `PeriodicTrigger` or a composite such as `QOSControl` in the case of instantiating `CompositeType`). An adaptive element instance is an instance of `AdaptiveElementInstance` type from the instance meta-model and can be conveniently created by invoking `newInstance` method defined in the `AdaptiveElementType`.

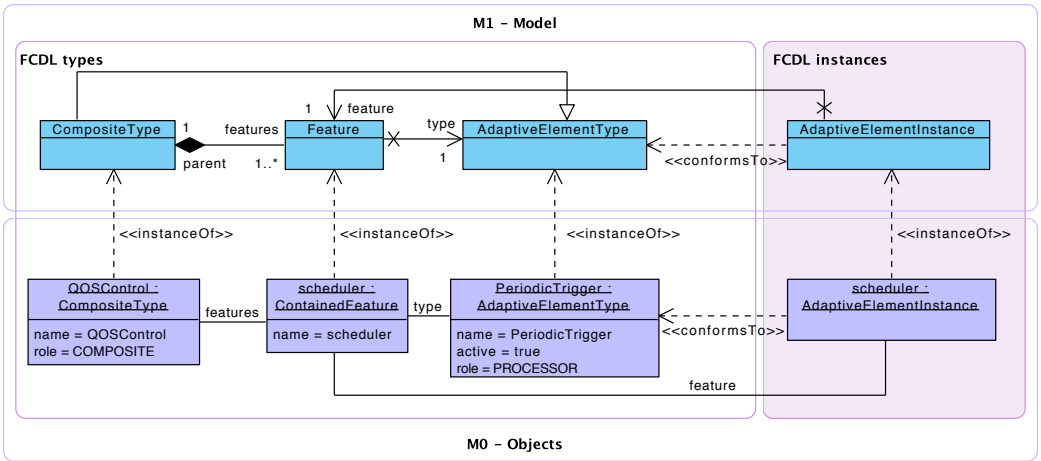


Figure 3.15: FCDL types and instances at two meta-modeling levels

Figure 3.15 illustrates the difference on the `PeriodicTrigger` example. It shows two meta-modeling layers *M1* and *M0* and the two FCDL packages `types` and `instances`. The top *M1* layer defines the FCDL modeling elements such as `AdaptiveElementType` and `AdaptiveElementInstance`. These are the elements of the FCDL abstract syntax. The bottom *M0* layer defines instances of the elements from the *M1* layer. This is the layer where the FCDL modeling happens. We start with the `PeriodicTrigger` type definition by creating an instance of `AdaptiveElementType`. Next, we need to create a composite

that will declare this adaptive element. We thus define the `QOSControl` composite with a feature called `scheduler`. By invoking the `newInstance` method on the `QOSControl` composite, we create an `AdaptiveElementInstance` of the `PeriodicTrigger` as defined by the `scheduler` feature.

An FCDL developer works exclusively with the type model defining adaptive element types. The instances are used by FCDL tools. When the model is used as an input to some tool such as code generator, it is instantiated and the tool works only with the instance graph.

Instantiation Essentially, instantiation is just a model-to-model transformation between the `types` and the `instances` meta-models, starting with the main composite types. The instance meta-model is shown in Figure 3.16 and the full types meta-model in Figure B.2.

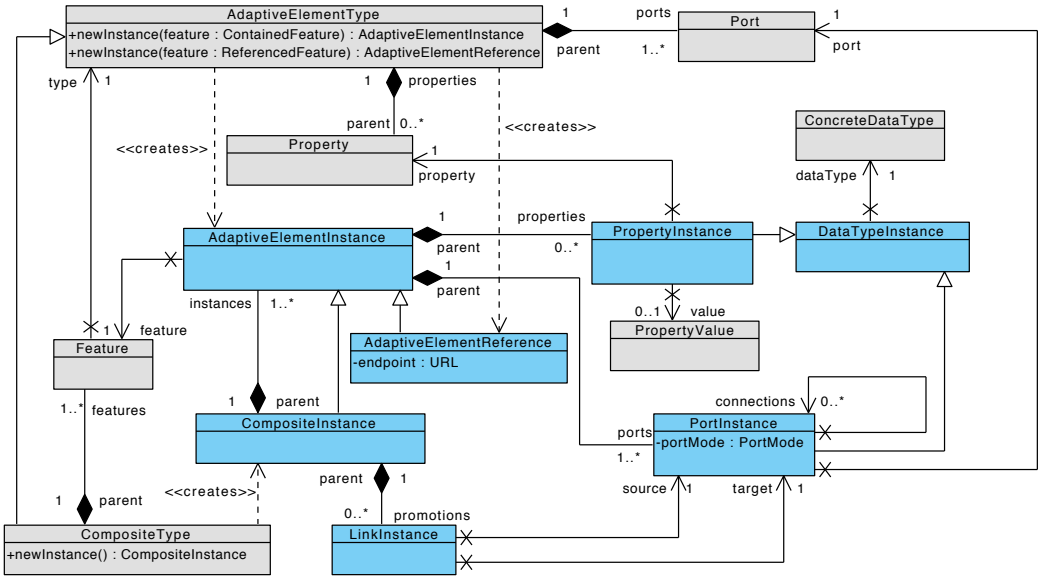


Figure 3.16: FCDL meta-model excerpt related to instances

By looking at the instance meta-model, the transformation is rather intuitive. Starting with a `CompositeType` it does the following mappings:

- `ContainedFeature` maps to either `AdaptiveElementInstance` or `CompositeInstance` depending on its type.
- `ReferencedFeature` maps to `AdaptiveElementReference`.
- `Port` maps to `PortInstance`.
  - agnostic ports mode are resolved based on the rules defined in Section 4.1.6,
  - if the data type is defined using data type parameter, a concrete data type is resolved by following the data argument definitions until its is defined by a concrete data type,
  - all connected ports are associated with the instance,
  - if port is promoted a new instance of `Promotion` is created.
- `Property` maps to `PropertyInstance`

  - if the data type is defined using data type parameter, a concrete data type is resolved in the same way as the port data type,
  - if there is a property value associated, it is also associated with the instance.
- `DataTypeParameters` and `DataTypeArguments` are resolved into `ConcreteDataTypes`.

3.3.8 Annotations

FCDL is used as an input to various model manipulation tools, for example, the already mentioned source code generator. For a tool to work properly, it might require some additional details about some aspects of the model. We have already mentioned this in the Section 3.3.2 where we discussed the data value serialization. This is a tool specific information and as such it should not be directly reflected in the meta-model. Instead, the FCDL meta-model uses *annotations* as a mechanism by which additional information can be attached to model elements. Figure 3.17 shows the meta-model excerpt related to annotations.

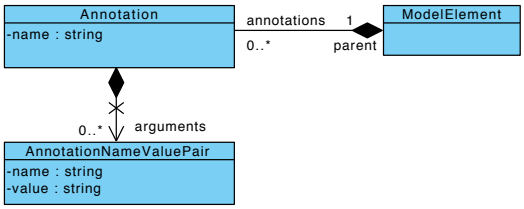


Figure 3.17: FCDL meta-model excerpt related to annotation

All model elements that subclass `ModelElement` class can be annotated. An `Annotation` is defined by its name and by a list of string name-value pairs. It is up to the tools to define the appropriate schema of the annotations it requires.

3.4 Summary

In this chapter we have presented the domain specific modeling language called *Feedback Control Definition Language* (FCDL), which is the backbone of our model-driven approach to external self-adaptive software systems development. The purpose of the modeling language is to expressively and concisely define architectures of external self-adaptive software systems through feedback control loops. These loops are reified as first-class entities at both design time and at runtime. The language is realized as an actor-oriented, component-based meta-model that provides flexible abstractions for modeling feedback control loops using hierarchically composed networks of collaborating actor-like entities called adaptive elements. FCDL is a strongly typed modeling language that includes first-class support for element composition, reflection and distribution across multiple runtime environments. The language abstract syntax is further associated with an informal graphical notation.

The language concepts were illustrated on a real-world self-adaptation scenario on QoS management control of web servers by content delivery adaptation.

CHAPTER 4

Modeling Feedback Control Architectures - Semantics

In the previous chapter we have described the concepts of actors, their compositions and communication channels that form the abstract syntax of our domain-specific modeling language, FCDL. However, this abstract syntax does not provide a clean semantics for this model. Therefore, in this chapter, we complement the model structure with operational rules that drive the execution of adaptive elements.

We start by describing FCDL semantics for adaptive element life-cycle, message passing and activations. In the second part we raise the level of abstraction on which the element interaction is defined by extending the semantics with a notion of interaction contracts that in turn enable better programming and verification support.

4.1 Model of Computation

The operational rules define the conditions for adaptive elements activation, as well as the overall interactions and communications between them, forming together a *Model of Computation* (MoC). A model of computation governs the semantics of the interactions [Brooks et al., 2010], addressing problems related to component activation, execution discipline and data propagation. It defines components in a concurrent system, their life-cycle and how they communicate and compute data.

Different systems need different models of computations. For example systems with continuous dynamics such as analog circuits and mechanical systems may desire MoC involving continuous time while signal processing systems will use synchronous dataflow MoC. Complex hybrid systems might even require multiple models of computation [Eker et al., 2003]. In FCDL we describe architectures of feedback control which are reactive systems responding to some stimulus. As we have shown in the motivating scenario (Section 3.2), the communication between feedback processes is a mixture of data-driven

and demand-driven communication. This type of interaction is generally referred to as a *push-pull* communication. A communication is *push* if it originates in a data producer (*e.g.* sensor) or *pull* if it is initiated by a data consumer (*e.g.* a processor) [Zhao, 2003]. The push-pull communication is studied in various systems, most notably in middleware services. The CORBA event service [Object Management Group, 2012a], Service Component Architecture [OASIS, 2007], but also Ptolemy 2 Component Interactions [Zhao, 2003] are all examples of systems that incorporate this communication style.

Our MoC has been inspired by the Ptolemy *Component Interactions* (CI). While conceptually both models of computation are close to one another, the way they are realized is very different. This is caused by the conceptual differences between these systems. Ptolemy models are executable. They can be directly simulated from within the model editor [Brooks et al., 2008]. An implementation of an actor is therefore tightly coupled with the actor framework provided by the Ptolemy environment. On the other hand, there is a loose coupling between Ptolemy actors and models of computation. Most of Ptolemy actors are polymorphic in the sense that they can operate within more than one model of computation. This together with the possibility of hierarchically composing different MoCs makes Ptolemy usable in a wide range of systems (including the above mentioned continuous dynamics and signal processing). FCDL models, on the other hand, are neither executable nor multi-domain. Adaptive elements are tightly coupled to one model of computation (defined by this section) that is tailored for defining feedback control for software systems. However, FCDL models are technologically agnostic. They are intended to be used as inputs to code generators that synthesize running system implementations based on the model definition for some particular runtime platform. Adaptive elements are thus loosely coupled with the actor framework that makes them operable within multiple runtime platforms.

In Ptolemy, a model of computation like CI is implemented by a software component called director that is responsible for ordering actors execution. A director uses multiple receivers that represent the communication links[1] defining the communication protocol. FCDL on the other hand leaves the ordering of adaptive element execution to the underlying actor framework dispatcher and it embeds the model of computation directly into the adaptive element. By using a delegate pattern [Grand, 1998] to separate the actual adaptive element behavior from the director it can abstract over different actor framework implementations.

4.1.1 Adaptive Element Director and Delegate

The reason for separating the concerns of general actor message processing and the specifics of adaptive elements, *i.e.*, life-cycle, activation and communication, is twofold: (1) to provide a higher-level abstraction encapsulating the structure and semantics defined in this chapter over the low-level actor message passing, and (2) to have the adaptive element implementations actor framework agnostic. For example, instead of directly sending mes-

[1] Communication channels in Ptolemy terminology

sages to actors, we use abstractions representing ports that take care of the data communication. This separation also simplifies adaptive element implementation and testing since it can be done in an isolation without dependency to any actor runtime. Moreover, because the element code does not directly depend on any particular actor framework library, it becomes reusable across multiple actor implementations within the same programming language. On the other hand, the MoC has to be implemented for each actor framework separately.

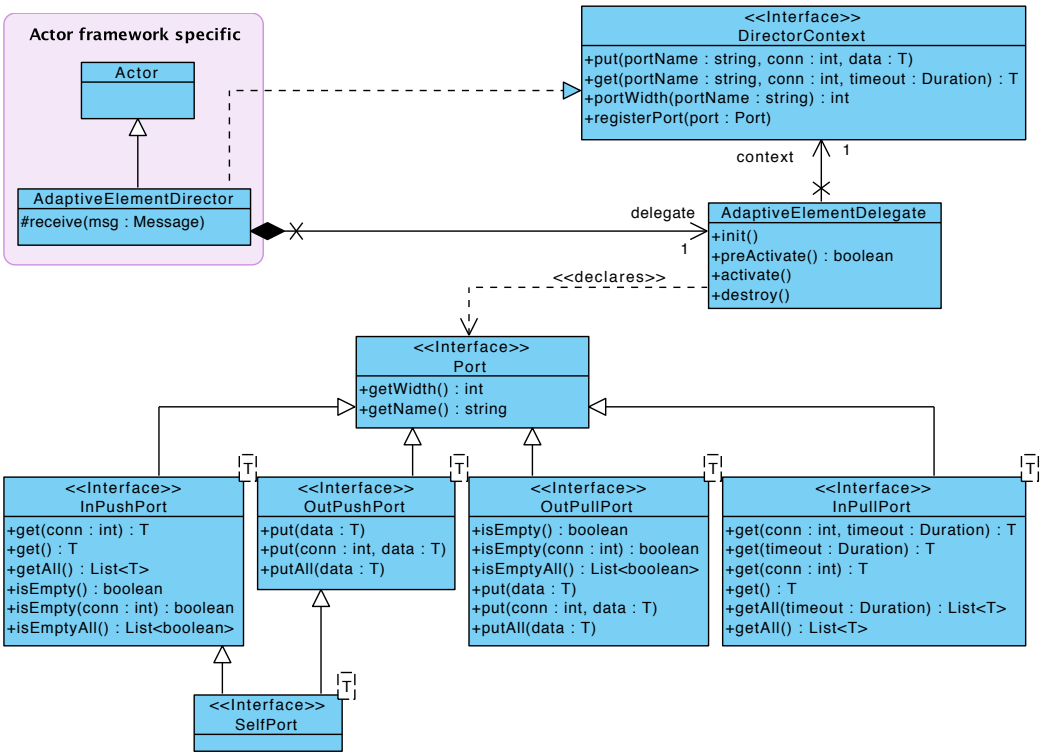


Figure 4.1: Adaptive element director and delegate

Figure 4.1 shows the separation between the actor part and the adaptive element specific part. An adaptive element director (`AdaptiveElementDirector`) bridges the underlying actor framework (represented by the `Actor` class[2]) and adaptive elements (`AdaptiveElementDelegate`). The director is responsible for managing the actor message queue and for the contained delegate life-cycle and activation. The delegate is where the actual state and behavior of an adaptive element is encapsulated and who performs the actual computation.

The `AdaptiveElementDelegate` is an abstract class. Concrete delegate classes should be synthesized from the adaptive element type definition from FCDL by a source code generator (*cf.* Section 5.2). A delegate communicates with its director through the `DirectorContext` interface. It is a façade [Gamma et al., 1994] that gives access to some

[2]Usually, in actor frameworks one creates a new actor by extending some actor base class or implements an actor interface.

common services provided by the framework and to the message passing mechanism used by ports. Next to the director context reference, an adaptive element delegate has the following methods:

`init()` is the initialization point of an adaptive element. It is invoked exactly once before any other method is invoked. It gives an opportunity to initialize any state variables an adaptive element might have. For active element, this is the place where event handlers are registered.

`destroy()` is the opposite of the `init` method. It is called once, but this time, at the end of the element life-cycle. It gives the element opportunity to clean up its state, unregisters any event handlers and release any resources it might own.

`preActivate()` method is called before adaptive element activation to check activation preconditions. It returns a boolean indicating whether the activation should proceed or not. This method provides a mechanism for the delegate to postpone its activation, for example until all the input ports receive data.

`activate()` represents the main point of execution. Essentially, it is responsible for getting values from the input ports, computing one or more results using these values and finally sending them to the connected elements through the output ports. Every adaptive element must implement this method.

We have already shown a part of the `PeriodicTrigger` implementation in Listing 3.1. In Listing 4.1 we extend the example with the activation method implementation and demonstrates some of the API.

```
1  class PeriodicTriggerDelegate(val ctx: DirectorContext)
2   extends AdaptiveElementDelegate {
3
4   // using context to create ports
5   val selfport: SelfPort[Long] = new DefaultSelfPort(ctx)
6   val input: InPullPort[T] = new DefaultInPullPort(ctx, "input")
7   val output: OutPushPort[T] = new DefaultOutPushPort(ctx, "output")
8
9   /** Adaptive element activation */
10  def activate {
11    logger debug s"Activated at: {selfport.get}"
12
13    Option(input.get) match {
14      case Some(data) => output put data // non-null data
15      case None => log info "No data available on the input port" // null
16    }
17  }
18 }
```

Listing 4.1: Example of adaptive element delegate

This code snippet declares a delegate class by extending the `AdaptiveElementDelegate` abstract class with the `DirectorContext` as the primary constructor parameter (lines 1 and 2). Next (lines 5-7), using the director context, the element's ports are defined. In this

case we use default implementations, but a developer can create custom port implementations (*cf.* below). The activation method (line 10) behavior is rather intuitive. It simply checks whether the received data contains some value (non-null) and if so, it forwards them to whatever element is connected to the push output port.

4.1.2 Message Passing

In FCDL, the communications originate in ports. At the type level, a port can be configured in one of the three modes: *push*, *pull* or *agnostic*. For now we will only consider push and pull modes as the agnostic mode eventually maps to one of the former two (the details of this mapping are described in Section 4.1.6). The push mode represents an interaction that is initiated by the data provider while the pull mode interaction originates at the data receiver. A connection is formed at the composite level by linking two ports, a source output port and a target input port. Based on the connected port modes, a connection can be either push (connecting two push ports) or pull (connecting two pull ports). We restrict the model to only allow connections between ports configured in the same mode. The very same restriction is done by the Ptolemy component interaction MoC. The reason is that connecting a push output port to a pull input port indirectly implies using a queue that acts as a storage component. Analogically connecting a pull output port to a push input requires to use a scheduler mediating the communication by periodically pulling and pushing the input and output ports. In FCDL, these interactions are intended to be explicitly modeled in the architecture in order to properly define the storage and the trigger mechanisms. For example, in the running example we explicitly place a scheduler (`PeriodicTrigger`) that periodically pulls the system utilization value from the monitor into the controller.

Ports Messages are sent and received via ports. However, ports never communicates directly with the connected peers. Instead, they proxy the communications through the director context. For example, invoking the `put` method on the `OutPushPort` will in turn invoke the `put` method defined in the `DirectorContext`. This interface is implemented by the adaptive element's director who is responsible for composing the message with the given data, locating the appropriate target element reference, and directing the message to it. Similarly, the director is also responsible for decoding an incoming message and placing the data to the appropriate port before activating the delegate. The concrete implementation of these operations are actor framework dependent.

The input push port (`InPushPort`) is basically a queue that stores the pushed value. During the delegate activation it may or may not dequeue the value and use it for the computation. If it does not, the value stays there. Default port implementation use regular unbounded FIFO queue, but developers can define a custom implementation (*e.g.* using a fixed size head dropping queue). The port offers a method `isEmpty` to check whether it contains a value or not. This is useful for input data synchronization when, for example, an activation should only proceed when two or more input ports have received data.

Multiports By default, one output port can be connected to only one input port and vice-versa. A port that needs to support more than one connection has to be explicitly defined as a *multiport*. In this case a port has an access to the underling ordered set of its connections and can distinguish between them using an index. The connections are indexed from 0 based on the connection order. All the port methods therefore take an argument that specifies which communication link should be used. For convenience, the port interface also provides methods that use implicitly the 0^{th} link and methods that executes over all connections (denoted by the `All` suffix). The current number of links can be obtained by invoking the `getWidth` port method. This number can potentially change during the execution since the model is dynamic and its structure can change at runtime (*cf.* Section 3.3.5). Again this method is just a proxy to the director context `portWidth` since it is the director which handles port connections.

Message Priority In the Section 3.3.5 we have mentioned message priorities. The messages sent from the provided ports, *i.e.*, from provided sensors and effectors have higher priority over the messages sent from the regular ports. This is to give higher significance to the reflection over the regular behavior.

Queue and Scheduler A connection is only allowed between ports configured in the same mode. In order to connect ports of the opposite modes we need to explicitly employ a queue or a scheduller. A queue is a passive processor that has a push input port and a pull output port. It enqueues all the inputs requests and dequeues them on demand made by pull requests on its output port. A queue therefore acts as a data storage between an element with a push output port and an element with a pull input port. There can be many different queue implementations. For example, the `Accumulator` from the running example is a queue that also processes the input data, in this case summing them together.

A schedule is an active processor with a pull input port and a push output port. It actively pulls data from its input port according to some scheduling policy and pushes it to its output. A scheduler is usually used to control the timing of requesting and distributing data within the feedback loop. It acts as a mediator between an adaptive element with a pull output port and an adaptive element with a push input port. For example, the `PeriodicTrigger` from the running scenario is a generic timer based scheduler.

4.1.3 Push Communication

Push communication is realized using the *fire and forget* message passing. In this mode, messages are sent asynchronously and the message producers do not expect any reply from the message consumer. Figure 4.2 illustrates the push communication.

The figure contains two adaptive elements an active sensor $\mathcal{A}$ and a processor $\mathcal{B}$ that are connected through push ports `outA` and `inB`. These elements could represent for example the `accessLog` and the `accessLogParser` from the running scenario. At some point, the event handler that is associated with the active sensor executes and requests the element to activate by pushing data d to its `selfport`. The actor framework is responsible

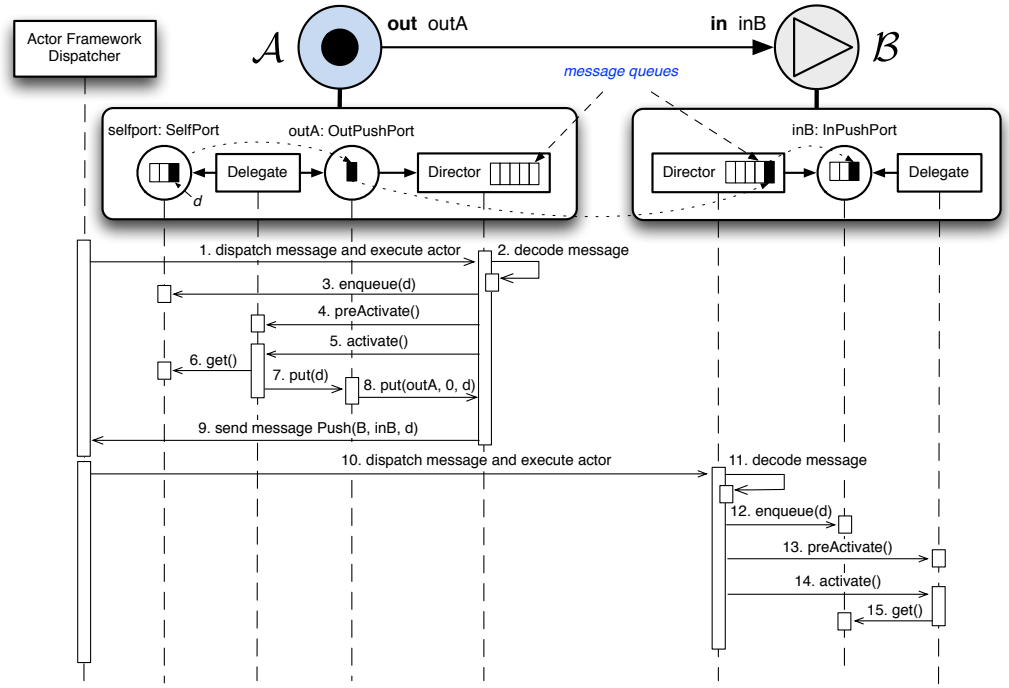


Figure 4.2: Push communication. Dash lines represents components time lines. Arrow represents methods calls and message passing.

for dispatching the message to the correct actor message queue (sometimes also called mailbox) and executing it at some convenient time (1.). The actual execution ordering is decided solely by the actor framework dispatcher. Usually, an actor framework creates a thread pool to execute as many actors concurrently as soon as they receive messages. Upon execution a director first decodes the received message and based on its type it executes the proper behavior (2.). In this case it is a push message and the director extracts the associated data from the message and enqueues it to the appropriate port, which is in this case is the `selfport` (3.). Next, it executes the `preActivation` method to check with the delegate if all preconditions are met, so the element can be consequently activated by invoking the delegate `activate` method (5.). In this example, the delegate simply collects the data from the self port (6.) and forwards it to its output port `outA` (7.).

The `outA` port handles the message sending through the director context by invoking the `put` method (8.). The second argument 0 in the `put` method call indicates that the first communication link should be used. The director encapsulates the given data d into a `Push` message and sends it to the appropriate actor reference representing the connected adaptive element (9.). This message will consequently activate $\mathcal{B}$ (10.). Similarly to the work that has been done by the $\mathcal{A}$ director, the $\mathcal{B}$ director also first decodes the message (11.), enqueues the data in the `inB` port (12.), checks the activation precondition (13.) and activates the $\mathcal{B}$ delegate (14.). Finally, the $\mathcal{B}$ delegate obtains the data d by dequeuing its input port `inB` (15.).

4.1.4 Pull Communication

The pull communication represents demand-driven interaction and it is realized by *send and receive* message passing. Unlike the push *fire and forget*, in this mode, the message producer expects a reply from the target and it will wait for it. While this might give an ``*illusion*'' of a synchronous communication, it is not since all messages in the actor model are sent asynchronously. The message is also sent asynchronously just like in the push mode, but additionally, it includes a reference to an actor to whom the message recipient should reply. This new destination is an adhoc created actor solely for the purpose of handling one particular reply and it will be destroyed as soon as it receives it. The received reply is made available through a *future*, an object representing potential reply from the destination [Baker and Hewitt, 1977].

There are at least three cases in which it might happen for the replay to never occur. (1) because, there are no guaranteed message delivery, the original request message or the eventual reply might get lost (particularly when the communicating elements are remotely distributed), (2) the target element crashed during the computation, or (3) the target element took too long for it to finish the computation[3]. Therefore there has to be always a timeout associated to each of the send and receive message call, terminating the adhoc actor and raising an exception in the calling thread so it is not blocked forever and resources are not leaked. The timeout can be set globally or individually using the `timeout` parameter of the `get` methods defined in the `InPullPort` interface.

It is important to realize that by using timeouts, a computation may be terminated before it finishes. For example, let us consider the running example. Instead of defining the `accessLog` sensor that actively pushes the newly added log messages, we could have just make a passive sensor that upon a request rereads the log file from the last known position. A potential problem[4] could occur during heavy load when there is lot of traffic and therefore lot of new lines in the log file. With long delays between the consecutive rereads of the log file, the pull message might eventually timeout before the sensor or the connected parser are done. That is why, it is always good to minimize the size of a pull chain (the number of consecutive pull requests).

Furthermore, there are some performance implications of using the pull communication. Each of the actor has to have a guard that keeps track of when it times out. Also, because waiting for the reply will effectively block the executing thread, the thread pool should be sufficiently sized otherwise a dead lock occurs. Therefore, in general, the push style of communication should used where possible.

4.1.5 Element Activation

In general, an actor can be activated if its mailbox contains at least one message. The ordering of the activations is determined by the actor framework dispatcher. In FCDL, a

[3] This case is rather a consequence of employing timeouts for the pull communication.

[4] This is unlikely to happen since the access log file parsing is very fast unless the file is located on a distributed file system in a saturated network.

message can only be sent by an adaptive element and therefore for a system to do something, there always has to be at least one active element. Once an active element is activated it consequently pulls or pushes data to its connected peers, which in turn causes their peers to activate and so on and so forth. Eventually, all defined elements should be activated at some point.

Definition 1 (Adaptive Element Activation) *An adaptive element $\mathcal{A}$ can be activated by either a push request over any of its push input ports, or a pull request over any of its pull output ports.*

Based on these rules, FCDL model can be checked that all defined elements will be eventually active. An adaptive element is eventually active if

1. it is an active actor (an active actor can send a message to itself through the implicit push input *selfport*), or
2. it has a pull output port connected to an eventually activated element, or
3. it has a push input port connected to an eventually activated element.

4.1.6 Agnostic Port Mode

The main motivation behind an agnostic port mode is to allow to define adaptive elements that can be used in both push and pull communication modes. For example, in the running scenario we might need to further stabilize the outputs from the utilization monitor and from the controller (*cf.* Figure 4.3). One way of doing so is to let the data pass through an additional processor, a moving average filter (`MovingAverage`), which computes a moving average value of a fix data series subset.

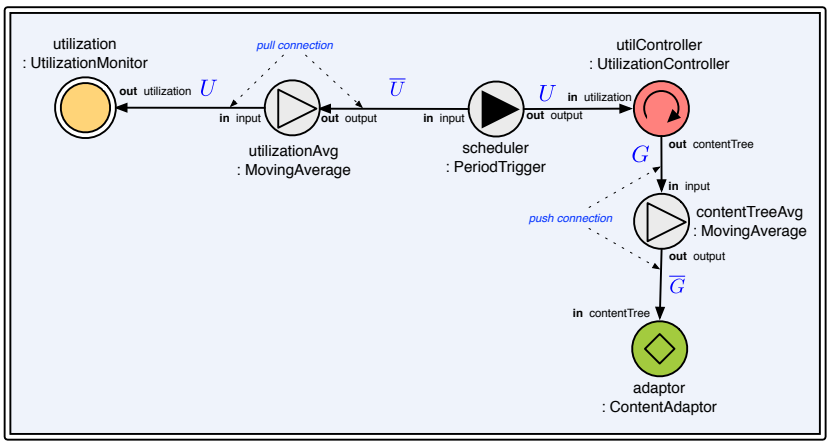


Figure 4.3: Example of an agnostic port mode. The `MovingAverage` has both ports configured in an agnostic mode and therefore it can be used in both pull and push communication modes.

If we tried to define such a filter, we would quickly realize a limitation in FCDL. The problem is that the data from the utilization monitor are pulled while the data from the

controller are pushed. With only a push mode and pull mode we would therefore need to define two different moving average processors, one usable for push connections and the other for pull connections.

This is clearly not a good strategy as it results in lots of duplications, violating our software engineering requirements. The moving average is an example of a set of adaptive elements that are also referred to as *transformers*. They take an input data, modify it in some way, and produce output data regardless whether the data are pushed or pulled.

A solution to this lies in the *agnostic* port mode. An agnostic port can be connected to either push or pull ports. The actual mode is determined when the port is connected, considering that connected ports must be configured in the same mode (*cf.* Section 4.1.2).

This impose several restrictions on using agnostic ports: (1) All agnostic ports within an adaptive element must eventually resolve into the same communication mode. (2) All agnostic input ports are considered to be synchronized. The element is only activated when all inputs received data. In the pull mode it means that the element director pulls all inputs before it executes the element. In the push mode, the director waits for all inputs to receive data and only then it tries to activate the element. (3) Adaptive elements with agnostic ports cannot be active, cannot have push input ports nor pull output ports.

4.2 Interaction Contracts

FCDL models an architecture of a feedback control system. In general architecture models are used for two main engineering concerns: for statical analysis and for mapping the architecture into an implementation [Oreizy et al., 1998]. A key element in both concerns is to have enough details about the data and control-flow interactions between the components in the architectural description. The level of details determines the amount of possible implementation guidance and the amount of programming and verification support that will be available.

In this section we extend the operational rules defined in above model of computation with more precise semantics that in turn will enable better programming and verification support.

4.2.1 Motivation

In the running example we used two elements to sum the number of requests and the size of responses. These processors simply accumulate the data received on their `input` port, returning the sum of all inputs once they are pulled on their `sum` output port. Let us now consider a more sophisticated[5] `Accumulator` as depicted in Figure 4.4

It works as follows: when it receives data on its `input` port, it pushes to its `output` port the input value plus the sum of all the input values it has received since the last time the `reset` port was triggered, similarly, when pulled on the `sum` port, it returns the sum of all

[5]Inspired by the Ptolemy 2 Accumulator actor http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.1/ptII/doc/codeDoc/ptolemy/actor/lib/Accumulator.html

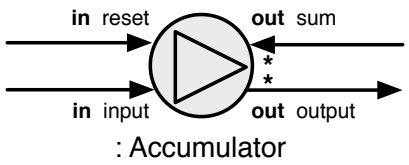


Figure 4.4: Improved `Accumulator` processor. The `sum` is a pull input and `output` is a push output multiport.

the input values since the last reset, and finally receiving any data on its `reset` port, sets the current accumulated value back to 0.

The above description makes the element interactions rather intuitive. However, for example the fact that every time an input is received data will be pushed over the output port is not explicitly stated in the architecture, but only informally expressed in the documentation. Therefore, in more complex cases or with less rigid documentation, this may potentially lead to different interpretations and incompatible implementations. Furthermore, since there is only one activation method per adaptive element, the control flow has to be manually coded. As a consequence, the element interactions becomes an integral part of its implementation, reducing the possibility of formal analysis.

For example, let consider a possible implementation of the accumulator in Scala from Listing 4.2.

```
val input: InPushPort[Long]
val output: OutPushPort[Long]
val reset: InPushPort[Any]
val sum: OutPullPort[Long]

var value = 0L

def activate() {
  if (!input.isEmpty) {
    // activated by a push on input
    value += input.get
    output send value
  } else if (!reset.isEmpty) {
    // activated by a push on reset
    reset.get
    value = 0
  } else if (!sum.isEmpty) {
    // activated by a pull on sum
    sum send value
  } else {
    throw new IllegalStateException("Invalid execution")
  }
}
```

Listing 4.2: Example of an `Accumulator` adaptive element implementation without an interaction contract

While the code is still quite concise, there are two main issues with it. First, if we look solely at the interface, it is a black box and there is no way to tell which of the ports will be used. Second problem is that a developer has to manually follow the data flow and synchronize the inputs in the case data from multiple input ports are needed at the same

time. In particular, a special attention should be paid to the lines 15 and 21. On the line 15 a developer has to make sure that the value from the port is consumed even though it is not used for any computation as the `reset` port acts merely as a trigger. On the line 21 one has to handle the control flow getting out of what is expected by throwing an exception. This is particularly important in cases when the flow is more complicated.

In short, the problem is that the architecture as it is, is underspecified. A similar issue of an architecture underspecification is discussed by Cassou *et al.* [Cassou et al., 2011] for SCC systems (*cf.* Section 2.2.2). To address the architecture underspecification, they prose to enrich SCC architecture descriptions by annotating components with *interaction contracts* that precise their interactions. We extend this notion and make it applicable to our model, *i.e.*, to precisely specify all possible interactions between a hierarchically organized networks of multiple-input, multiple-output adaptive elements. Concretely, our extension to the original interaction contracts includes support for:

- *Components with multiple output ports.* Adaptive elements, unlike the SCC components, may have multiple outputs.
- *Multiports.* Any adaptive element port can be linked to multiple targets which is not the case of the SCC model.
- *Composites.* FCDL support hierarchical composition of adaptive elements while the SCC components are all defined in a single scope.
- *Optional interaction contracts.* Since the SCC components have only one output, all interaction contracts are compulsory and must be satisfied within an architecture. On the contrary, adaptive elements can define an optional interaction (*e.g.* the reset functionality in the `Accumulator`).
- *Interaction contract completion verification.* Complementing the optional interaction contracts with a formal verification that checks whether all required ports are connected.

The use of interaction contracts brings two main advantages to the previously defined model of computation:

1. By using interaction contracts we can assert certain architectural properties such as consistency (*cf.* Section 4.2.5), determinacy (*cf.* Section 4.2.6), and completeness (*cf.* Section 4.2.7).

   The different possible activations are no longer hidden in the documentation. Instead, they are clearly visible in the interface of the adaptive element and therefore amenable to automatized analysis and verification (*cf.* Section 5.3).

2. Interaction contracts enable to write more concise and natural implementation (*cf.* Section 4.2.8). For example, the `Accumulator` can be rewritten as:

```
var value = 0L

def onInput(input: Long): Long = {
  value += input
  value
}
```

```
def onReset(reset: Any) {
  value = 0
}

def onSum(): Long = value
```

3. Moreover, interaction contracts allow the code generating facility can be greatly improved. Instead of generating only one `activate` method for the entire adaptive element behavior, with interaction contracts it can generate precise methods for each element activation (*cf.* Section 4.2.8). This way, the generated code is both *prescriptive* (guiding the developer) and *restrictive* (limiting the developer to what the architecture allows). It is arguably easier to implement the above three methods than to write the complete implementation of the `activate` method as shown in Listing 4.2.

Before we can introduce the interaction contracts we need to formally define some aspects of the FCDL meta-model that have been described in the previous sections. Concretely, we use sets and tuples to formally define the main FCDL elements: an adaptive, a port and a composite. These prerequisites (Section 4.2.2) are then followed by the interaction contract definition for adaptive elements (Section 4.2.3) and for composite (Section 4.2.4). Based on these three sections we formulate the architecture properties such as consistency (*cf.* Section 4.2.5), determinacy (*cf.* Section 4.2.6), and completeness (*cf.* Section 4.2.7). Finally, we present the interaction contract denotation and mapping into activation method signatures (Section 4.2.8).

4.2.2 Prerequisites

Throughout this section we will use the following notations and definitions. The lower case letters (e) and singular symbols (var) refer to scalar variables. Upper case letters (S) or plural symbols ($vars$) refer to sets of variables. Bold text (**true**) represents constants. For example $\exists e \in S$ such that $e =$ **true**. The `monospaced` font is used for functions and references to elements defined in FCDL.

All sets are finite and possibly empty unless explicitly stated that a non-empty set ($\neq \emptyset$) is required. We use $\bigcup M$ to denote a union of a set whose elements are sets themselves: $x \in \bigcup M \iff \exists A \in M$ such that $x \in A$. We use $\_\_$ as a placeholder that can replace a variable, a tuple or a set so to match any value. For example if $T = (\{\textbf{true}, \textbf{false}\}, \{1, 2\})$ then $(\_\_; 1)$ matches tuples $(\textbf{true}, 1)$ and $(\textbf{false}, 1)$. We use $\perp$ to represent an ``*absent*'' value, a **null**, or to indicate no result value in a function call (a `void`).

In Section 3.3.2 we discussed the notion of a type system. For concrete examples in this section we will use the Scala type system[6] and reference its basic types such as `Int`, `Long` and `Double`.

Using this notation, we formally define the main elements of the FCDL instance model (*cf.* Section 3.3.7) that are necessary for describing interaction contracts. Some of the following definitions could have been written in a more compact way. However, since they

[6] As we have discussed in the Section 3.3.2, FCDL does not define a type system on its own but rather uses a type system from the target programming language.

also denote structural invariants of the FCDL meta-model, the additional verbosity simplifies their eventual mapping into the implementation and, although subjectively, makes them easier to parse and express as structural constraints (*cf.* Section 7.2).

Definition 2 (Adaptive Element) *An adaptive element is a tuple*

$$\mathcal{A} = (I, O, \alpha, self)$$

where I is a set of input ports, O is a set of output ports, α is an interaction contract and $self$ is a port representing an active adaptive element self port or $\perp$ in the case $\mathcal{A}$ is passive. An adaptive element must always define at least one input or output port, $|I \cup O| \geq 1$.

The `Accumulator` adaptive element is therefore defined as[7]:

$$(\{\texttt{reset}, \texttt{input}\}, \{\texttt{sum}, \texttt{output}\}, \_, \perp)$$

Definition 3 (Port) *A port is a tuple*

$$p = (type, mode, multi, prov, t, conns)$$

where $type = \{\textbf{in}, \textbf{out}, \textbf{self}\}$ is the type of the port, $mode = \{\Uparrow, \Downarrow, \Updownarrow\}$ indicates the mode of the port to be respectively push, pull or agnostic, $multi = \{\textbf{true}, \textbf{false}\}$ indicates whether the port is a multi port or a single port, $prov = \{\textbf{true}, \textbf{false}\}$ distinguishes between a regular port and a provided port, t is the data type of values accepted by the port and conns is a set of connected ports.

For example, the `Accumulator` adaptive element from Figure 4.4 has following ports:
`reset` $= (\textbf{in}, \Uparrow, \textbf{false}, \textbf{false}, \texttt{Any}, \_)$, `input` $= (\textbf{in}, \Uparrow, \textbf{false}, \textbf{false}, \texttt{Long}, \_)$,
`sum` $= (\textbf{out}, \Downarrow, \textbf{true}, \textbf{false}, \texttt{Long}, \_)$, and `output` $= (\textbf{out}, \Uparrow, \textbf{true}, \textbf{false}, \texttt{Long}, \_)$.

For ports we define a function `typeof` (p) that for a given port p returns its data type t:

$$\texttt{typeof}\,(p) \quad = \quad \begin{cases} t \text{ if } p \text{ is a port} \\ \textbf{T} \text{ if } p \text{ is an input multi port and } \textbf{T} \text{ is vector of } t \end{cases}$$

Given t_1 and t_2 to be data types, $t_1 \leq t_2$ denotes that t_1 is assignable from t_2 (t_2 conforms to t_1)[8] and $t = t_1 \cup t_2$ defines a data type union t such that $t \leq t_1 \wedge t \leq t_2$. For example, in Scala such type union is the smallest common supertype of t_1 and t_2.

A function `port_conns` (p) for a given port $p = (type, mode, multi, prov, t, conns)$ returns the set of the ports $conns$ that are connected to the port p or $\emptyset$ if there are no ports connected to p.

Furthermore, through the text we use $\Uparrow$ to represent *push mode*, $\Downarrow$ for *pull mode* and $\Updownarrow$ for *agnostic mode*. These symbols can be used as superscripts to sets of ports in which case they act as predicates selecting only ports configured in the given mode. Let O be a set of ports, $O^{\Uparrow} = \{p | p \in O \wedge \texttt{ispush}\,(p)\}$ where `ispush` (p) is a predicate that is **true** if

[7] The `Accumulator` interaction contract will be defined later in this section
[8] Scala rules for type conformance are defined in the §3.5.2 of Scala language specification [Odersky, 2011].

p is in push mode. For example, let I and O be respectively the sets of input and output ports of the `Accumulator` processor, $I^{\Uparrow} = \{\texttt{reset}, \texttt{input}\}$, $I^{\Downarrow} = \emptyset$, $O^{\Uparrow} = \{\texttt{output}\}$ and $O^{\Downarrow} = \{\texttt{sum}\}$.

Definition 4 (Composite) *A composite Γ is an adaptive element $(I, O, \alpha, \perp)$ that additionally defines two sets A, P, where A is a non-empty set of adaptive elements contained in Γ and P is a set of port promotions.*

In the scope of a given composite Γ, we define a function `is_promoted` (p) that return **true** if a given port p has been promoted to one of the composite ports or **false** otherwise.

4.2.3 Definition and Properties of an Interaction Contract

The objective of an interaction contract is to describe the allowed interactions of an adaptive element. First we need to define what interactions activate an adaptive element. Once an element is activated, it might need to request additional data through one or more of its pull input ports and finally, depending on its intention and its internal state, it may or may not push the results of the computation to one or more of its push output ports. To formally specify this interaction information we introduce the concept of a *basic interaction contract* for an adaptive element.

Definition 5 *(Basic Iteration Contract) Let $\mathcal{A}$ be an adaptive element $(I, O, \alpha, self)$, a basic iteration contract α associated with an adaptive element $\mathcal{A}$ is a tuple*

$$\alpha = \langle\, A; R;\, E \,\rangle$$

where A, R, E represents respectively the activation condition, the data requirements and the data emissions that are defined as follows:

- *Activation condition*

$$A = \Uparrow (I_1, \ldots, I_n) \mid self \mid \Downarrow (out)$$

  represents when an adaptive element is executed:

  - $\Uparrow (I_1, \ldots, I_n)$ *where I_i is either a unit set containing a single push input port $in \in I^{\Uparrow}$ or a disjunction of such ports $in_1 \vee \ldots \vee in_n,\ in_i \in I^{\Uparrow}$, corresponding to data received on these sets of ports I_i. When I_i is a disjunction of ports then any port receiving data can be used.*
  - *self corresponds to an activation caused by the self port and it is only applicable if $\mathcal{A}$ is active. In such a case it must declare an interaction contract that has self in the activation condition.*
  - $\Downarrow (out)$ *where $out \in O^{\Downarrow}$ corresponds to a pull request on a pull output port out. A pull requests always returns data to the calling adaptive element.*

- *Data requirements*

$$R = \Downarrow (in_1, \ldots, in_n)$$

  where $in_i \in I^{\Downarrow}$ and $1 < n \leq |I^{\Downarrow}|$ represent ports that may be pulled during the execution.

– *Data emissions*

$$E = \Uparrow (out_1, \ldots, out_n, out_{n+1}?, \ldots, out_{n+m}?)$$

where $out_i \in O^{\Uparrow}$ *and* $0 < n + m \leq |O^{\Uparrow}|$ *indicates to what output ports the result of the adaptive element execution will be emitted. The* ? *marks an optional data emission. When* $A = \Downarrow (out), out \in O^{\Downarrow}$*, then a value is always emitted to the corresponding pull output port* out*, answering the pull request.*

Using interaction contracts we can precisely specify interactions of an adaptive element. For example, the basic interaction contracts of the adaptive elements from the running example (*cf.* Figure 3.3) are shown in the Figure 4.5 (`Accumulator`'s interaction contract will be discussed later).

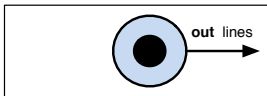
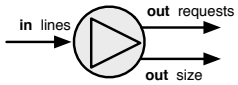
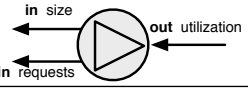
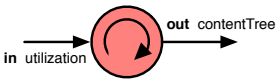
	FileTailer	$\langle\, self; \emptyset; \Uparrow (\texttt{lines}) \,\rangle$
	AccessLogParser	$\langle\, \Uparrow (\texttt{lines}); \emptyset; \Uparrow (\texttt{requests}, \texttt{size}) \,\rangle$
	ContentAdaptor	$\langle\, \Uparrow (\texttt{input}); \emptyset; \emptyset \,\rangle$
	LoadMonitor	$\langle\, \Downarrow (\texttt{utilization}); \Downarrow (\texttt{requests}, \texttt{size}); \emptyset \,\rangle$
	PeriodicTrigger	$\langle\, self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?}) \,\rangle$
	UtilizationController	$\langle\, \Uparrow (\texttt{utilization}); \emptyset; \Uparrow (\texttt{contentTree}) \,\rangle$

Figure 4.5: Interaction contracts of the running example adaptive elements

In the rest of this subsection we will detail the properties of interaction contracts. Concretely, we will describe interaction contract composition, the differences between input synchronization and disjunction, default interaction contracts for elements that do not have any explicit contract assigned, provide some additional details regarding to agnostic ports, multi ports and provided ports, and finally we will introduce interaction contract inference for composites.

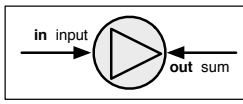
Contract composition An adaptive element can have multiple interaction contracts associated where the different activation conditions lead to different behaviors. For example `Accumulator` outlined in the motivation (*cf.* Section 4.2.1) defines three different behaviors depending on which port activates it.

Definition 6 (Contract Composition) *Two or more basic interaction contracts* $\alpha_1, \ldots, \alpha_n$ *can be combined together using the* $\|$ *operator such as* $\alpha_1 \,\|\, \cdots \,\|\, \alpha_n$ *where* $n \geq 2$*. Additionally, a*

basic interaction contract can be marked with ? indicating an optional interaction, but there have to be always at least one compulsory contract.

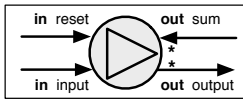
For example, the two `Accumulator` adaptive elements have following interaction contracts:

`Accumulator` (from 3.3):



$\langle \Uparrow (\texttt{input}); \emptyset; \emptyset \rangle \parallel \langle \Downarrow (\texttt{sum}); \emptyset; \emptyset \rangle$

`Accumulator` (from 4.4):



$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle \parallel \langle \Uparrow (\texttt{reset}); \emptyset; \emptyset \rangle ? \parallel \langle \Downarrow (\texttt{sum}); \emptyset; \emptyset \rangle$

An interaction contract can be considered as a set $\alpha = \{\alpha_i, \ldots, \alpha_n\}$ of basic interaction contracts α_i where $n \geq 1$. This is also how it is represented in the meta-model (*cf.* page 87). However, the notation with $\parallel$ operator is preferred in text as it is more compact and expressive. Furthermore, we define an interaction contract merge operator that is used for interaction contract inference in composite types (*cf.* Section 4.2.4)

Definition 7 (Interaction Contract Merge Operator) *Merge can be used both as a binary and a unary operator. In the binary form, a merge of two basic interaction contracts $\alpha = \alpha_1 \otimes \alpha_2$ where $\alpha_1 = \langle A_1; R_1; E_1 \rangle$ and $\alpha_2 = \langle A_2; R_2; E_2 \rangle$ is an interaction contract $\alpha = \langle A_1 \cup A_2; R_1 \cup R_2; E_1 \cup E_2 \rangle$. The unary form of the merge operation is*

$$\otimes\alpha \quad = \quad \begin{cases} \alpha_1 \otimes \cdots \otimes \alpha_n & \text{if } \alpha = \alpha_1 \parallel \cdots \parallel \alpha_n \\ \alpha & \text{if } \alpha \text{ is a basic interaction contract} \end{cases}$$

Input Synchronization An activation condition $A = \Uparrow (I_1, \ldots, I_n)$ in the interaction contract represents data synchronization. An element is then only activated when $\forall I_i \in A, \exists in \in I_i$ such that in received data.

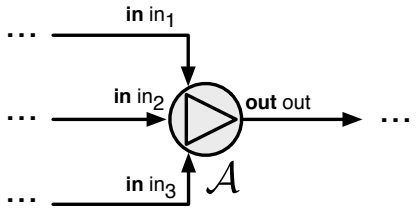


Figure 4.6: Example of input synchronization

Let consider a passive adaptive element $\mathcal{A}$ with three input push ports and one push output port (*cf.* Figure 4.6). An interaction contract

$$\langle \Uparrow (\texttt{in}_1, \texttt{in}_2, \texttt{in}_3); \emptyset; \Uparrow (\texttt{out}) \rangle$$

will only activate $\mathcal{A}$ if values have been pushed to all of its input ports and the activation will happen as soon as the last port is pushed. This activation will correspond to a method:

```
def onIn1_In2_In3(in1: T, in2: T, in3: T): T
```

On the other hand an interaction contract

$$\langle \Uparrow (\mathtt{in}_1); \emptyset; \Uparrow (\mathtt{out}) \rangle \parallel \langle \Uparrow (\mathtt{in}_2); \emptyset; \Uparrow (\mathtt{out}) \rangle \parallel \langle \Uparrow (\mathtt{in}_3); \emptyset; \Uparrow (\mathtt{out}) \rangle$$

will activate $\mathcal{A}$ any time there has been a data pushed on any of its input ports. This corresponds to three different activation methods:

```
def onIn1(in: T): T
def onIn2(in: T): T
def onIn3(in: T): T
```

Input Disjunction An activation condition A of an interaction contract can contain a disjunction of ports. For example, let considered $\mathcal{A}$ from Figure 4.6 above. An activation condition $\Uparrow (\mathtt{in}_1 \vee \mathtt{in}_2, \mathtt{in}_3)$ will active $\mathcal{A}$ in two cases:

1. when data has been received on $\mathtt{in}_1$ and on $\mathtt{in}_3$, or
2. when data has been received on $\mathtt{in}_2$ and on $\mathtt{in}_3$.

It is important to mention that in the case like this, timing plays an important role. For example, if $\mathtt{in}_1$ receives data before $\mathtt{in}_2$ and $\mathtt{in}_2$ receives data before $\mathtt{in}_3$, then the adaptive element will be activated using values from ports $\mathtt{in}_1$ and $\mathtt{in}_3$. On the one hand this introduces a non-determinism into the model since there are no guarantees about message delivery timeliness. On the other hand, by defining activation using port disjunction it should be clear that it does not matter which of the port has received the data first.

Let us consider again the contract $\langle \Uparrow (\mathtt{in}_1); \_; \_ \rangle \parallel \langle \Uparrow (\mathtt{in}_2); \_; \_ \rangle \parallel \langle \Uparrow (\mathtt{in}_3); \_; \_ \rangle$ activating $\mathcal{A}$ any time data is received on any of its input ports. Similarly, a condition $\langle \Uparrow (\mathtt{in}_1 \vee \mathtt{in}_2 \vee \mathtt{in}_3); \_ ; \_ \rangle$ will active $\mathcal{A}$ any time one of the input ports $\mathtt{in}_1, \ldots, \mathtt{in}_3$ receives data. However, the crucial difference here is that in the former case there is a different behavior associated with each of the three interaction contracts corresponding to three different activation methods (*cf.* above). In the latter case, on the other hand, the same behavior is always executed regardless of which port port activated the element. There is therefore only one activation method:

```
def onIn123(in: T): T
```

Default Interaction Contracts An adaptive element that is not a composite and that does not define an interaction contract is implicitly assigned a default one upon instantiation. The contract is inferred according to the activation rules defined by the model of computation, definition 1 (*cf.* Section 4.1.5).

Definition 8 (Default Interaction Contract) *Let $\mathcal{A}$ be a non-composite adaptive element $(I, O, \alpha, self)$, a default interaction contract α is a set of basic interaction contracts:*

$$\left\{\langle \Uparrow (in_i); R;\ E \rangle \mid in_i \in I^{\Uparrow}\right\} \cup \left\{\langle \Downarrow (out_i); R;\ E \rangle \mid\ out_i \in O^{\Downarrow}\right\} \cup \begin{cases} \langle self; R;\ E \rangle & \text{if } self \neq \bot \\ \emptyset & \text{otherwise} \end{cases}$$

where $R = \Downarrow (I^{\Downarrow})$ *and* $E = \Uparrow (out_1?, \ldots, out_n?),\ out_i \in O^{\Uparrow},\ n = |O^{\Uparrow}|$

Agnostic Port Mode In the Section 4.1.6 we have discussed that the actual mode for an agnostic port is determined at the point when the port is connected and that all agnostic ports are configured in the same mode. Therefore, there are two possible interaction contracts to be associated with an adaptive element that has one or more agnostic ports.

Definition 9 (Interaction Contract for an Adaptive Element with Agnostic Ports) *Let $\mathcal{A}$ be an adaptive element $(I, O, \alpha, \bot)$ with one or more agnostic ports. Its interaction contract must be*

- *For the push mode*
  $\alpha = \langle \Uparrow (in_i, \ldots, in_n); \Downarrow (I);\ E \rangle$ *where* $in_i \in \Updownarrow I, n = |\Updownarrow I|$ *and* $\Updownarrow O \subseteq E \wedge \forall out_j \in O^{\Uparrow} : (out_j \in E \vee out_j? \in E)$.
- *For the pull mode*
  $\alpha = \langle \Downarrow (out_i); \Downarrow (I);\ E \rangle \parallel \cdots \parallel \langle \Downarrow (out_n); \Downarrow (I);\ E \rangle$ *where* $in_i \in \Updownarrow I, n = |\Updownarrow I|$ *and* $\Updownarrow O \subseteq E \wedge \forall out_j \in O^{\Uparrow} : (out_j \in E \vee out_j? \in E)$.

The actual mode is determined at the instantiation. However, for activation method generation the push mode is assumed since it makes the implementation more concise.

Multiports There is one exception to what has been defined for the multiports in the model of computation (Section 4.1.2 on page 70). When using interaction contracts, input multiports are implicitly synchronized and the resulting data type is a vector $\mathbf{T}$[9] of the port data type t.

Provided Sensors and Effectors Technically provided sensors and effectors are ports as well and thus they are also part of the interaction contracts. If a behavior represented by a basic interaction contract α needs to broadcast a state change using a provided sensor ps, it has to declare this fact explicitly in the emission part E of the interaction contract α. Similarly, for each provided effector, there has to exist a basic interaction contract, that actives the element upon a push request over the provided port. There are no special constraints, but since the reason for defining a provided effector is to enable to adjust the behavior of an adaptive element, the activation condition should not involve any other

[9] For example in Scala this vector is represented by `scala.collection.immutable.Seq[T]`

port. Moreover, there should not be any data requirements and the only possible emissions should be directed to provided sensors. Therefore, a basic interaction contract for a provided effector pe should match $\langle \Uparrow (pe); \emptyset; \Uparrow (ps_1, \ldots, ps_n, ps_{n+1}?, \ldots, ps_m?) \rangle$ where ps_i is a provided sensor of the adaptive element defining pe.

For example the `PeriodicTrigger` from Figure 3.11 defining a provided effector `setPeriod` has its interaction contract: $\langle self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output}?) \rangle \parallel \langle \Uparrow (\texttt{setPeriod}); \emptyset; \emptyset \rangle$. In case it also defined a provided sensor `period` allowing other elements to be notified every time the scheduling changes, the second basic interaction contract would then be $\langle \Uparrow (\texttt{setPeriod}); \emptyset; \Uparrow (\texttt{period}?) \rangle$[10].

4.2.4 Interaction Contracts for Composites

A composite is also an adaptive element and as such an interaction contract is also associated to it, enabling to fully assert the promised architecture properties. A composite, however, does not provide any particular behavior on its own. It acts merely as a mediator routing data from the promoted ports to the contained adaptive elements. Its interaction contract therefore does not have to be explicitly specified, but instead, it is inferred from the interaction contracts of the contained elements.

Let us consider the composite shown in Figure 4.7. It presents one composite containing three adaptive elements with the following interaction contracts:

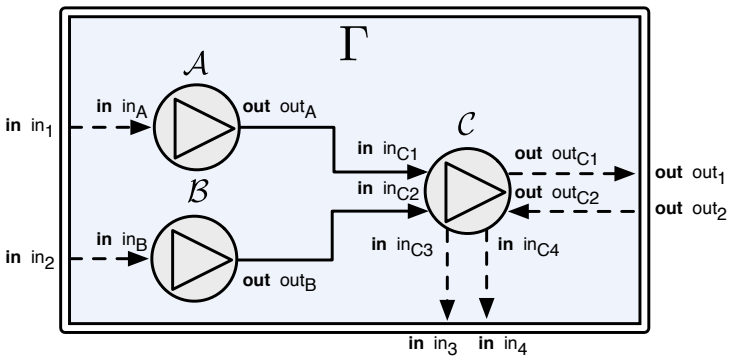


Figure 4.7: Example composite for the interaction contract inference illustration.

$$
\begin{aligned}
\alpha_{\mathcal{A}} &= \langle \Uparrow (\texttt{in}_\texttt{A}); \emptyset; \Uparrow (\texttt{out}_\texttt{A}) \rangle \\
\alpha_{\mathcal{B}} &= \langle \Uparrow (\texttt{in}_\texttt{B}); \emptyset; \Uparrow (\texttt{out}_\texttt{B}) \rangle \\
\alpha_{\mathcal{C}} &= \langle \Uparrow (\texttt{in}_{\texttt{C1}}, \texttt{in}_{\texttt{C2}}); \Downarrow (\texttt{in}_{\texttt{C3}}); \Uparrow (\texttt{out}_{\texttt{C1}}) \rangle \parallel \langle \Downarrow (\texttt{out}_{\texttt{C2}}); \Downarrow (\texttt{in}_{\texttt{C4}}); \emptyset \rangle
\end{aligned}
$$

The first part of the interaction contract is the activation condition. This can contain either one or more push input ports, a self port or a pull output port. Since the depicted composite defines a pull output port ($\texttt{out}_2$) we can infer that it has to have one basic interaction contract $\langle \Downarrow (\texttt{out}_1); \_ ; \_ \rangle$. For the two of its push input ports ($\texttt{in}_1, \texttt{in}_2$), however, there are two alternatives. They can be either part of the same contract $\langle \Uparrow (\texttt{in}_1, \texttt{in}_2); \_ ; \_ \rangle$

[10] The emission is optional since it should only broadcast a change if the newly set period is different from the old one, which does not always have to be the case.

or two different ones $\langle \Uparrow (\mathtt{in}_1); \_ ; \_ \rangle \parallel \langle \Uparrow (\mathtt{in}_2); \_ ; \_ \rangle$. Simply by looking at the composite structure or even at the interaction contracts associated to these ports we cannot determine the exact activations, neither can we assess which of the pull input ports ($\mathtt{in}_3, \mathtt{in}_4$) belongs to which activation nor what activation uses the push output port ($\mathtt{out}_1$).

To infer interactions, we need to explore the complete graph of connected ports, finding all execution paths within the composite that interact with any of the promoted ports. An execution path is an ordered set of basic interaction contracts that starts in some promoted port from which it follows data propagation among adaptive elements.

We now illustrate the algorithm used for the composite interaction contract inference using the above composite (*cf.* Figure 4.7). The formal definition of the algorithm is given in Figure 4.8 and a concrete implementation in Scala is listed in Appendix B.4.

The first step in the contract inference is to compute the execution paths starting from any of the composite ports that can be part of an interaction contract activation condition (step 1 on line 6 of the algorithm). In our example, these ports are $\mathtt{in}_1$, $\mathtt{in}_2$ and $\mathtt{out}_2$. We start with first port, $\mathtt{in}_1$. This port is promoted to the $\mathcal{A}$ input $\mathtt{in}_\mathtt{A}$ that is associated with a single interaction contract $\alpha_\mathcal{A}$. This contract thus becomes the first contract in the execution path. Next, we need to explore all interaction contracts of the adaptive elements that are connected to any of the ports associated in this interaction contract $\alpha_\mathcal{A}$. We do that by visiting all the communication links of all the ports that are part of the contract, but the ones we have already visited ($\mathtt{in}_\mathtt{A}$). There is only one other port, $\mathtt{out}_\mathtt{A}$, associated with $\alpha_\mathcal{A}$ and thus we continue the traversal to all its connections. The only connection from $\mathtt{out}_\mathtt{A}$ goes to the $\mathtt{in}_{\mathtt{C1}}$ that belongs to $\mathcal{C}$. This element interaction contract is a composition of two basic interaction contracts. We have to therefore find out what contracts are associated with the port we are currently visiting. In this case it is the first basic interaction contract $\langle \Uparrow (\mathtt{in}_{\mathtt{C1}}, \mathtt{in}_{\mathtt{C2}}); \Downarrow (\mathtt{in}_{\mathtt{C3}}); \Uparrow (\mathtt{out}_{\mathtt{C1}}) \rangle$ that associates $\mathtt{in}_{\mathtt{C1}}$ port. However, in general it can be more than one contract since we can be visiting a push output port which can be part of the multiple data emissions. At this point, the execution path so far is:

$$\{\langle \Uparrow (\mathtt{in}_\mathtt{A}); \emptyset; \Uparrow (\mathtt{out}_\mathtt{A}) \rangle, \langle \Uparrow (\mathtt{in}_{\mathtt{C1}}, \mathtt{in}_{\mathtt{C2}}); \Downarrow (\mathtt{in}_{\mathtt{C3}}); \Uparrow (\mathtt{out}_{\mathtt{C1}}) \rangle\}$$

There are three remaining ports to visit, $\{\mathtt{in}_{\mathtt{C2}}, \mathtt{in}_{\mathtt{C3}}, \mathtt{out}_{\mathtt{C1}}\}$. Both $\mathtt{in}_{\mathtt{C3}}$ and $\mathtt{out}_{\mathtt{C1}}$ are promoted ports and thus are not connected to any elements within the composite. The $\mathtt{in}_{\mathtt{C2}}$ leads to $\mathcal{B}$ element's port $\mathtt{out}_\mathtt{B}$ associated with the $\alpha_\mathcal{B}$ interaction contract. By adding this contract to the execution path, the exploration is complete since there are no more ports to visit (in_B is a promoted port). The complete execution path that starts in the $\mathtt{in}_1$ port is thus composed of these interaction contracts:

$$\alpha_{\mathtt{in}_1} = \{\langle \Uparrow (\mathtt{in}_\mathtt{A}); \emptyset; \Uparrow (\mathtt{out}_\mathtt{A}) \rangle, \langle \Uparrow (\mathtt{in}_{\mathtt{C1}}, \mathtt{in}_{\mathtt{C2}}); \Downarrow (\mathtt{in}_{\mathtt{C3}}); \Uparrow (\mathtt{out}_{\mathtt{C1}}) \rangle, \langle \Uparrow (\mathtt{in}_\mathtt{B}); \emptyset; \Uparrow (\mathtt{out}_\mathtt{B}) \rangle\}$$

In the very same way we can obtain the execution path for the $\mathtt{in}_2$ port:

$$\alpha_{\mathtt{in}_2} = \{\langle \Uparrow (\mathtt{in}_\mathtt{B}); \emptyset; \Uparrow (\mathtt{out}_\mathtt{B}) \rangle, \langle \Uparrow (\mathtt{in}_{\mathtt{C1}}, \mathtt{in}_{\mathtt{C2}}); \Downarrow (\mathtt{in}_{\mathtt{C3}}); \Uparrow (\mathtt{out}_{\mathtt{C1}}) \rangle, \langle \Uparrow (\mathtt{in}_\mathtt{A}); \emptyset; \Uparrow (\mathtt{out}_\mathtt{A}) \rangle\}$$

and for the $\mathtt{out}_2$ port:

$$\alpha_{\mathtt{out}_2} = \{\langle \Downarrow (\mathtt{out}_{\mathtt{C2}}); \Downarrow (\mathtt{in}_{\mathtt{C4}}); \emptyset \rangle\}$$

```
1:  function composite_interaction_contract(Γ)                    ▷ Γ is a composite
2:      I⇑ ← push input ports of Γ
3:      O⇓ ← pull output ports of Γ
4:      S ← self ports ports of Γ
5:      exec_paths ← ∅                                  ▷ The set of execution paths
6:      for all port in I⇑ ∪ O⇓ ∪ S do                  ▷ Step 1: Find all execution paths
7:          S ← S ∪ exec_path (port)
8:      end for
9:      contracts ← ∅                                   ▷ The set of merged execution paths
10:     for all path in exec_paths do                   ▷ Step 2: Merge execution paths
11:         contracts ← contracts ∪ (⊗path)
12:     end for
13:     remove duplicates in contracts                  ▷ Step 3: Remove contract duplicates
14:     ics ← ∅                                         ▷ The final set of interaction contracts for Γ
15:     for all α in contracts do                       ▷ Step 4: Maps to promoted ports
16:         A, R, E ← α                                 ▷ α = ⟨ A; R; E ⟩
17:         A′ ← {{promoted_port (p) |p ∈ A_i} |A_i ∈ A}
18:         A′ ← {A_i|A_i ∈ A′ ∧ |A_i| > 0}              ▷ Remove empty port disjunctions
19:         R′ ← {promoted_port (p) |p ∈ R}
20:         E′ ← {promoted_port (p) |p ∈ E}
21:         ics ← ⟨ A′; R′; E′ ⟩
22:     end for
23:     return ⊗ics
24: end function
```

```
1:  function exec_path(p, U = ∅)  ▷ p is a port to visit, U is a set of already visited ports
2:      U ← U ∪ {p}                                     ▷ mark p as visited
3:      α ← ⊗ the interaction contracts associated with p
4:      A, R, E ← α                                     ▷ α = ⟨ A; R; E ⟩
5:      V ← (⋃ A ∪ R ∪ E) \ U                           ▷ ports to visit
6:      C ← {α}
7:      for all v in V do
8:          for all conn in port_conns (v) do
9:              C ← C ∪ ⋃ exec_path (q, U)
10:         end for
11:     end for
12:     return C
13: end function
```

```
1:  function promoted_port(p)                                          ▷ p is a port
2:      return ← the target of the promoted port p or ⊥ if p is not a promoted port
3:  end function
```

Figure 4.8: Interaction contract inference algorithm for composites

Each of the execution paths defines what ports are used to activate elements, what are their data requirements and data emissions. From the composite perspective, we are not interested in the individual contracts, but rather in the global view, *i.e.*, what are all the ports involved in the element activations, data requirements and emissions. Therefore we merge (*cf.* Definition 7) all individual contracts within an execution path together into a

single basic interaction contract (step 2 on line 10):

$$\begin{aligned}
\otimes\alpha_{\mathtt{in}_1} &= \langle \Uparrow (\mathtt{in_A}, \mathtt{in_{C1}}, \mathtt{in_{C2}}, \mathtt{in_B}); \Downarrow (\mathtt{in_{C3}}); \Uparrow (\mathtt{out_A}, \mathtt{out_{C1}}, \mathtt{out_B}) \rangle \\
\otimes\alpha_{\mathtt{in}_2} &= \langle \Uparrow (\mathtt{in_B}, \mathtt{in_{C1}}, \mathtt{in_{C2}}, \mathtt{in_A}); \Downarrow (\mathtt{in_{C3}}); \Uparrow (\mathtt{out_B}, \mathtt{out_{C1}}, \mathtt{out_A}) \rangle \\
\otimes\alpha_{\mathtt{out}_2} &= \langle \Downarrow (\mathtt{out_{C2}}); \Downarrow (\mathtt{in_{C4}}); \emptyset \rangle
\end{aligned}$$

The only difference between $\alpha_{\mathtt{in}_1}$ and $\alpha_{\mathtt{in}_2}$ contract is the order of ports, but eventually they represent the very same interaction and thus only one is needed. The next step is therefore to remove all duplicate interaction contracts that only differ in the port ordering (step 3 on line 13). Finally (step 4 on line 15), we have to map the ports back to their promotion where applicable and discard the ports that are not promoted (*cf.* Figure 4.9). Again, we only care about the ports that are promoted since these are the ports available on the outside of the composite. By composing these two resulting interaction contracts we derive the final

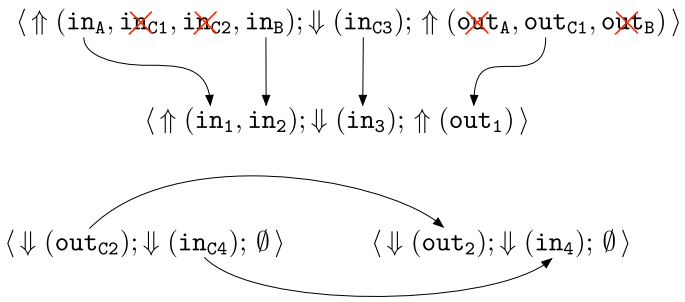


Figure 4.9: Example of mapping interaction contract ports into the composite ports

contract for the composite: $\alpha_\Gamma = \langle \Uparrow (\mathtt{in_1}, \mathtt{in_2}); \Downarrow (\mathtt{in_3}); \Uparrow (\mathtt{out_1}) \rangle \parallel \langle \Downarrow (\mathtt{out_2}); \Downarrow (\mathtt{in_4}); \emptyset \rangle$

Using this algorithm, we can infer the interaction contracts of the composites presented in the running example (*cf.* Figure 3.8). The resulting contracts are shown in Figure 4.10.

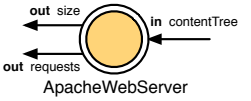
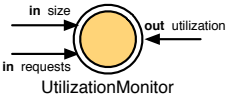
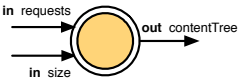
	$\langle self; \emptyset; \Uparrow (\mathtt{requests}) \rangle \parallel \langle self; \emptyset; \Uparrow (\mathtt{size}) \rangle \parallel \langle \Uparrow (\mathtt{contentTree}); \emptyset; \emptyset \rangle$
	$\langle \Uparrow (\mathtt{requests}); \emptyset; \emptyset \rangle \parallel \langle \Uparrow (\mathtt{size}); \emptyset; \emptyset \rangle \parallel \langle \Downarrow (\mathtt{utilization}); \emptyset; \emptyset \rangle$
	$\langle \Uparrow (\mathtt{requests}); \emptyset; \emptyset \rangle \parallel \langle \Uparrow (\mathtt{size}); \emptyset; \emptyset \rangle \parallel \langle self; \emptyset; \Uparrow (\mathtt{contentTree}) \rangle$

Figure 4.10: Interaction contracts of the composites from the running example

Representation of Interaction Contract in the Meta-Model Figure 4.11 shows the interaction contract representation in the FCDL meta-model. Both, the `AdaptiveElementType` and `AdaptiveElementInstance` contains a reference to `InteractionContract`. However,

at the type definition, the association is optional and a default interaction contract will be used during an instantiation if an element does not explicitly define one. The class `InteractionContract` represents a basic interaction contract and a composite contract is represented as a set.

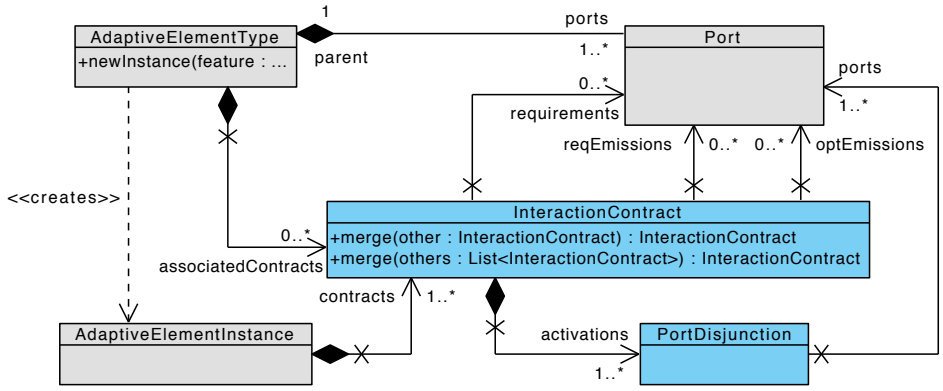
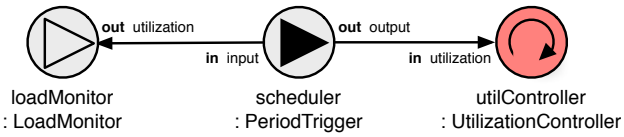


Figure 4.11: FCDL meta-model excerpt related to interaction contracts

4.2.5 Consistency

Besides main composites, adaptive elements are never used alone, instead, they are always part of an assembly in some composite Γ. By associating interaction contracts to adaptive elements, we are therefore not only defining the adaptive elements interactions themselves, but also implying certain interactions requirements for the other elements within Γ in order to have consistent assembly.

For example, let consider element interactions from the following excerpt of the running scenario taken from Figure 3.3:



For the `scheduler` to be able to pull data from its `input` port, it must define a contract with data requirements $R = \Downarrow$ (`input`). This implies that the connected element `loadMonitor` must have one of its basic interaction contracts to be $\langle \Downarrow$ (`utilization`); $\_$; $\_ \rangle$ since `utilization` output port is connected to the `PeriodicTrigger` `input`. Similarly, to push data to its `output` port, the `scheduler` defines an emission $E = \Uparrow$ (`output`?) in its associated contract implying that the connected element `controller` has a contract $\langle \Uparrow$ (`utilization`); $\_$; $\_ \rangle$. Since `scheduler` has neither push input ports, nor pull output ports it must be an active processor defining one basic interaction contract with activation condition $A = self$. More formally we define these rules as *contract consistency* and *composite consistency*.

In the scope of a given composite Γ, we define a function `port_contracts` (p) that for a given port p returns the interaction contract α associated to an adaptive element $\mathcal{A}$ that defines the port p as its either input or output port.

Definition 10 (Contract Consistency) *Given a composite Γ an interaction contract α associated with an adaptive element defined in Γ is consistent if one of the following conditions is satisfied:*

- *if α is a basic interaction contract $\alpha = \langle\, A; R;\, E \,\rangle$ then:*
  - *pull inputs consistency*
    $\forall in_i \in R \;\wedge\; \neg$`is_promoted` $(in_i)\,, \forall out_j \in$ `port_conns` (in_i) :
    $\exists \alpha' \in$ `port_contracts` (out_j) *such that* $\alpha' = \langle \Downarrow (out_j); \_ ; \_ \,\rangle$
  - *pull outputs consistency*
    if $A = \Downarrow (out)$ *then*
    $\forall in_i \in$ `port_conns` $(out) \;\wedge\; \neg$`is_promoted` (in_i) :
    $\exists \alpha' \in$ `port_contracts` (in_i) *such that* $\alpha' = \langle \_ ; R'; \_ \,\rangle \wedge in_i \in R$
  - *push outputs consistency*
    $\forall out_i \in E \;\wedge\; \neg$`is_promoted` $(out_i)\,, \forall in_j \in$ `port_conns` (out_i) :
    $\exists \alpha' \in$ `port_contracts` (in_j) *such that*
    $\alpha' = \langle\, A'; \_ ; \_ \,\rangle \wedge A' = \Uparrow (\_\,) \wedge in_j \in \bigcup A$
  - *push inputs consistency*
    if $A = \Uparrow (\_\,)$ *then*
    $\forall in_i \in \bigcup A \;\wedge\; \neg$`is_promoted` $(in_i)\,, \forall out_j \in$ `port_conns` (in_i) :
    $\exists \alpha' \in$ `port_contracts` (out_j) *such that*
    $\alpha' = \langle \_ ; \_ ;\, E' \,\rangle \wedge (out_j \in E' \vee out_j? \in E')$
- *if α is a composite interaction contract $\alpha = \alpha_1 \parallel \cdots \parallel \alpha_n$ then each α_i is considered individually.*

Definition 11 (Connected Optional Interaction Contract) *Within a given composite Γ an optional interaction contract $\alpha = \langle\, A; R;\, E \,\rangle?$ associated with some adaptive element defined in Γ is considered connected if $\exists p \in P$ such that* `port_conns` $(p) \neq \emptyset$ *where P is a set of ports defined as:*

$$P \;=\; \begin{cases} \bigcup A \cup R \cup E & \text{if } A = \Uparrow (\dots) \\ R \cup E & \text{if } A \text{ is } self \\ \{out\} \cup R \cup E & \text{if } A = \Downarrow (out) \end{cases}$$

Definition 12 (Composite Consistency) *A composite Γ is consistent if all compulsory and all connected optional interaction contracts associated with all adaptive elements defined in Γ are consistent.*

4.2.6 Determinacy

An interaction contract can be composed of one or more basic interaction contracts defining multiple activation condition for an adaptive element. It is important to make sure that these activation conditions do not interfere with one another and that a given data

flow always triggers only one basic interaction contract. An interference occurs for example if two or more interaction contracts shares the same push input port for its activation condition. In such a case the activation is not *deterministic* since it is not possible to say which contract should be executed when the data arrives. For example, considering the adaptive element $\mathcal{A}$ from Figure 4.6, an interaction contract

$$\langle \Uparrow (\mathtt{in}_1, \mathtt{in}_2, \mathtt{in}_3); \emptyset; \Uparrow (\mathtt{out}) \rangle \parallel \langle \Uparrow (\mathtt{in}_1); \emptyset; \Uparrow (\mathtt{out}) \rangle$$

is non-deterministic, because the two composed basic contracts shares the same push input port in their activation conditions.

Definition 13 (Contract Interference) *Two basic interaction contracts $\langle A; B; C \rangle$ and $\langle A'; B'; C' \rangle$ associated with $\mathcal{A}$ interfere with each other if $\bigcup A \cap \bigcup A' \neq \emptyset$*

Definition 14 (Contract Determinacy) *A contract composition $\alpha_1 \parallel \cdots \parallel \alpha_n$ is deterministic if each basic interaction contract α_i does not interfere with any of the others.*

Definition 15 (Composite Determinacy) *A composite Γ is deterministic if all interaction contracts of all adaptive elements defined in Γ are deterministic.*

4.2.7 Completeness

Interaction contracts also allow to verify whether an adaptive element interaction has been completely specified. For each port that has been defined in the adaptive element type, there exists an interaction contract that is using data from this port either for its activation, data requirements or data emissions.

Definition 16 (Contract Completeness) *Given an adaptive element $\mathcal{A} = \{I, O, \alpha, self\}$ an interaction contract $\alpha = \langle A_1; R_1; E_1 \rangle \parallel \cdots \parallel \langle A_n; R_n; E_n \rangle$ where $n \geq 1$ associated with $\mathcal{A}$ is complete if:*

$$I^{\Downarrow} = \bigcup_{i=1}^{n} R_i \ \wedge \ O^{\Uparrow} = \bigcup_{i=1}^{n} E_i \ \wedge \ I^{\Uparrow} \cup O^{\Downarrow} = \begin{cases} \bigcup_{i=1}^{n} A_i \backslash \{self\} & \text{if } self \neq \bot \\ \bigcup_{i=1}^{n} A_i & \text{otherwise} \end{cases}$$

4.2.8 Activation Methods and Adaptive Element Acts

The behavior of an adaptive element is a function that is executed in a reaction to data being received on the element input ports (function arguments) producing a result that is disseminated over the element output ports (function return values). An interaction contract give details on this function and therefore the denotational semantics of an interaction contract is of a function type. The interaction contract denotation enables to synthesize adaptive elements activation method signatures. Instead of having one activation method as is in the case of adaptive element delegate (*cf.* Section 4.1.1), we can now provide multiple methods for the fine-grained activations described by interaction contracts. Each basic interaction contract can be translated into a method that will be called

by the delegate at the appropriate time, when the activation condition is satisfied. To clearly separate this, we encapsulate all the interaction contracts activation methods in a separate class called *adaptive element act*. The adaptive element delegate `activate` method can then be automatically generated and the execution outsourced to the appropriate activation method in the new adaptive element act class.

Interaction Contract Denotation The mapping of an interaction contract into a function type is rather intuitive. An interaction contract $\alpha = \langle\, A; R;\, E \cup E? \,\rangle$ denotes a function $A \times R \times E? \to E$ where the E and $E?$ represents respectively the mandatory and optional data emissions. The activation condition A together with the data requirements R and optional data emissions $E?$ form the function parameters while the mandatory emissions E determine its return type. To indicate the optionality of the interactions for both the data requirements (pull inputs) and emissions (push output), we further define the following functions (*cf.* next paragraph for their concrete representations in Scala):

$$\begin{aligned} \texttt{push\_typeof}\,(p) &= \texttt{typeof}\,(p) \to () \\ \texttt{pull\_typeof}\,(p) &= () \to \texttt{typeof}\,(p) \end{aligned}$$

A special case is when an adaptive element is active. In this case the first function parameter is of type $\texttt{typeof}\,(self)$ where $self$ is the adaptive element self activation port (Definition 2). For example, an interaction contract $\langle \Uparrow (in_1); \Downarrow (in_2); \Uparrow (out_1, out_2, out_3?) \rangle$ denotes a function

$$\texttt{typeof}\,(in_1) \times \texttt{pull\_typeof}\,(in_2) \times \texttt{push\_typeof}\,(out_3) \to \texttt{typeof}\,(out_1) \times \texttt{typeof}\,(out_2)$$

and the interaction contract of the `PeriodicTrigger` $\langle\, self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output}?) \,\rangle$ denotes a function

$$\texttt{typeof}\,(self) \times \texttt{pull\_typeof}\,(input) \times \texttt{push\_typeof}\,(output) \to \bot$$

The complete denotation $[\![\alpha]\!]$ of an interaction contract α is summarized in Figure 4.14 on page 94.

Activation Methods Signatures Now we show an example of how the function types can be mapped into Scala methods. We use following interfaces to encapsulate the optional interactions such as data requirements and optional data emissions.

$$\begin{aligned} \texttt{typeof}\,(p) \to () \quad &\Rightarrow \quad \texttt{Push}[\texttt{typeof}\,(\texttt{p})] \quad \text{for optional data emission} \\ () \to \texttt{typeof}\,(p) \quad &\Rightarrow \quad \texttt{Pull}[\texttt{typeof}\,(\texttt{p})] \quad \text{for data requirements} \end{aligned}$$

The `Push` and `Pull` (*cf.* Figure 4.13) are interfaces representing specialized variants of the `OutPushPort` and `InPullPort` (*cf.* Section 4.1.2). The main difference is in handling multiports that are effectively transparent when using interaction contracts (*cf.* Section 4.2.3, page 83).

The last interaction contract example from the previous section $\langle \Uparrow (in_1); \Downarrow (in_2); \Uparrow (out_1, out_2, out_3?) \rangle$ maps to the following Scala activation method:

```
def activate(in1: R1, in2: Pull[R2], out3: Push[S3]): (S1, S2)
```

where R_1, R_2 and $S_1, \ldots, S_3$ are respectively the data types defined for the input ports `typeof` (in_1), `typeof` (in_2) and output ports `typeof` $(out_1), \ldots,$ `typeof` (out_3). The interaction contract associated with the `PeriodicTrigger` $\langle\, self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?}) \,\rangle$ maps to:

```
def activate(selfport: Long, input: Pull[T], output: Push[T]): Unit
```

The first argument represents data pushed through the `selfport` by the associated event handler (*cf.* Listing 3.1, line 12). The second argument, `input`, is a pull wrapper that when called pulls the data from the `input` port. Finally, the `output` parameter is a push wrapper for the optional emission over the `output` port. Since there are no required data emissions, the method returns nothing. The type argument T used in both wrappers suggests that `PeriodicTrigger` is a polymorphic adaptive element, *i.e.*, it defines a data type parameter of the same name (*cf.* Section 3.3.2). In Figure 4.2.8 we provide activation methods of all the elements participating in the running example.

Interaction Contract Acts Figure 4.12 shows the difference between the Scala user code needed for implementing the `Accumulator` processor from the beginning of this section (*cf.* Section 4.2.1) and the new code in the adaptive element act.

```
def activate() {
  if (!input.isEmpty) {
    // activated by a push on input
    value += input.get
    output send value
  } else if (!reset.isEmpty) {
    // activated by a push on reset
    reset.get
    value = 0
  } else if (!sum.isEmpty) {
    // activated by a pull on sum
    sum send value
  } else {
    throw new IllegalStateException("
        Invalid execution")
  }
}
```

(a) delegate activation method

```
def onInput(input: Long): Long = {
  value += input
  value
}

def onReset(reset: Any) {
  value = 0
}

def onSum(): Long = value
```

(b) act methods

Figure 4.12: Differences between adaptive element delegate and act activations

The more natural implementation in 4.12(b) is because we raised the level of abstraction. With interaction contract, it is the adaptive element delegate who is responsible for matching the port state with the appropriate activation denoted by the interaction contract activation condition. Furthermore, another advantage of using the interaction contract is that the generated interface is both prescriptive and descriptive. It guides the developer

while at the same time it only allows to perform permitted interactions. It does not provide any means to allow an interaction that has not been specified within the contract[11].

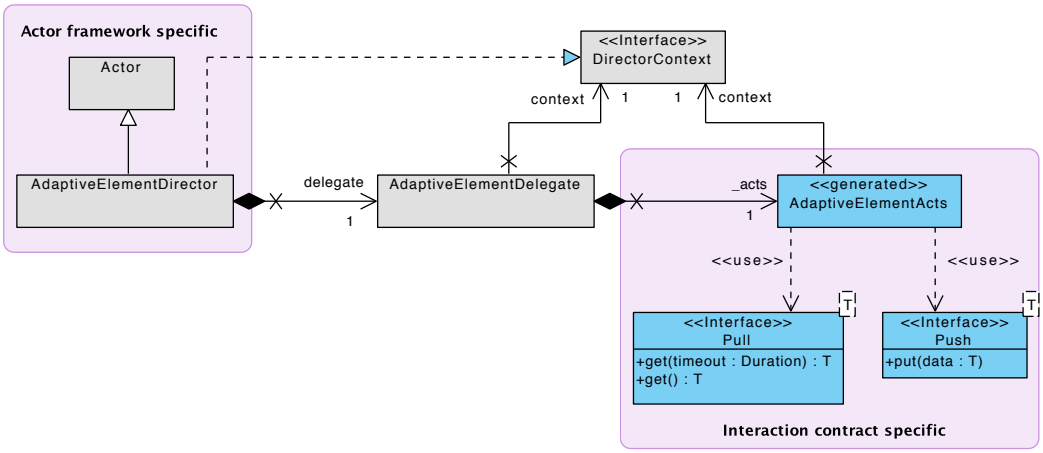


Figure 4.13: Interaction contract act

Figure 4.13 shows the relation between an adaptive element delegate and an adaptive element act synthesized from the element interaction contracts. Essentially, it is yet another delegation that raises the abstraction level on which the implementation is provided. Also, by further delegating the adaptive element life-cycle methods, we can fully generate the adaptive element delegate class so user only need to edit the adaptive element acts. Editing adaptive element act class should make the development better focused than writing the delegate code (*cf.* Figure 4.12) where one has to take care about the state flow manually.

[11] Even if a developer stored a reference to the given `Push` or `Pull` interaction and spawned a new thread that used them, there would not be any interaction as the underlying objects are detached from the ports after the activation method executes. However, there is no compilation time enforcement and only a runtime exception is thrown.

Denotation of a push activation.
Uses input ports disjunction for the function parameters representing the activation.

$$[\![\langle \Uparrow (I_1, \ldots, I_n); \Downarrow (in_1, \ldots, in_m); \Uparrow (out_1, \ldots, out_r, out_{r+1}?, \ldots, out_s?) \rangle]\!] = \prod_{i=1}^{n} \bigcup \{\texttt{typeof}\,(p) \,|p \in I_i\} \times \prod_{j=1}^{m} \texttt{pull\_typeof}\,(in_j) \times \prod_{l=r+1}^{s} \texttt{push\_typeof}\,(out_l?) \rightarrow \prod_{k=1}^{r} \texttt{typeof}\,(out_k)$$

Denotation of a self activation.
Similar to the push activation, but only uses data from the self port for the function parameters representing the activation.

$$[\![\langle self; \Downarrow (in_1, ..., in_m); \Uparrow (out_1, \ldots, out_r, out_{r+1}?, \ldots, out_s?) \rangle]\!] = \texttt{typeof}\,(self) \times \prod_{j=1}^{m} \texttt{pull\_typeof}\,(in_j) \times \prod_{l=r+1}^{s} \texttt{push\_typeof}\,(out_l?) \rightarrow \prod_{k=1}^{r} \texttt{typeof}\,(out_k)$$

Denotation of a pull activation.
Similar to the self activation, but only uses data from output pull port for the function parameters representing the activation.

$$[\![\langle \Downarrow (out); \Downarrow (in_1, ..., in_m); \Uparrow (out_1, \ldots, out_r, out_{r+1}?, \ldots, out_s?) \rangle]\!] = \texttt{typeof}\,(out) \times \prod_{j=1}^{m} \texttt{pull\_typeof}\,(in_j) \times \prod_{l=r+1}^{s} \texttt{push\_typeof}\,(out_l?) \rightarrow \prod_{k=1}^{r} \texttt{typeof}\,(out_k)$$

Denotation of an interaction contract composition.

$$[\![\alpha_1 \parallel \cdots \parallel \alpha_n]\!] = [\![\alpha_1]\!] \times \cdots \times [\![\alpha_n]\!]$$

Denotation of an optional interaction contract.

$$[\![\alpha?]\!] = [\![\alpha]\!]$$

Figure 4.14: Interaction contracts denotation. The first three denotations only differ in the first part of the function type that represents the activation parameters.

Adaptive Element	Interaction Contract / Method Signature
`FileTailer`	$\langle\, self; \emptyset; \Uparrow (\texttt{lines}) \,\rangle$
	`def activate(selfport: String): String`
`AccessLogParser`	$\langle\, \Uparrow (\texttt{lines}); \emptyset; \Uparrow (\texttt{requests}, \texttt{size}) \,\rangle$
	`def activate(lines: String): (Int, Long)`
`ContentAdaptor`	$\langle\, \Uparrow (\texttt{input}); \emptyset; \emptyset \,\rangle$
	`def activate(input: Double): Unit`
`LoadMonitor`	$\langle\, \Downarrow (\texttt{utilization}); \Downarrow (\texttt{requests}, \texttt{size}); \emptyset \,\rangle$
	`def activate(requests: Pull[Int],` `  size: Pull[Long]): Double`
`PeriodicTrigger`	$\langle\, self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?}) \,\rangle$
	`def activate(selfport: Long,` `  input: Pull[T], output: Push[T]): Unit`
	$\langle\, \Uparrow (\texttt{setPeriod}); \emptyset; \emptyset \,\rangle$
	`def onSetPeriod(input: Long): Unit`
`UtilizationController`	$\langle\, \Uparrow (\texttt{utilization}); \emptyset; \Uparrow (\texttt{contentTree}) \,\rangle$
	`def activate(utilization: Double): Double`
`Accumulator` (from 3.3)	$\langle\, \Uparrow (\texttt{input}); \emptyset; \emptyset \,\rangle$
	`def onInput(input: T): Unit`
`Accumulator` (from 4.4)	$\langle\, \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \,\rangle$
	`def onInput(input: T): T`
	(or in case when input is a multiport)
	`def onInput(input: Seq[T]): T`
	$\langle\, \Uparrow (\texttt{reset}); \emptyset; \emptyset \,\rangle$?
	`def onReset(reset: Any): Unit`
	$\langle\, \Downarrow (\texttt{sum}); \emptyset; \emptyset \,\rangle$
	`def onSum(): T`
`MovingAverage` (from 4.3)	$\langle\, \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \,\rangle$
	`def activate(input: T): T`

Figure 4.15: Examples of activation method signatures

4.3 Summary

In this chapter we have complemented FCDL description with semantics governed by a model of computation. It describes FCDL components behavior, *i.e.*, how they communicate and compute data. Then, we have presented the FCDL communication model that supports a mixture of data-driven and demand-driven communications (push-pull). In order to increase precision of adaptive element interactions and the level of abstraction on which it is described, the model of computation is further extended by the notion of interaction contracts (*cf.* Figure 4.16).

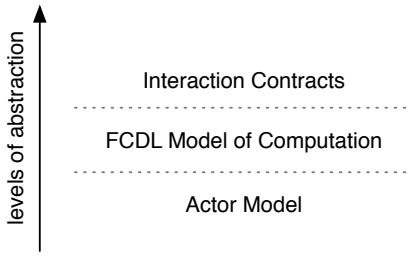


Figure 4.16: The different levels of FCDL semantic abstraction

The use of interaction contracts allows to assert certain architectural properties such as consistency, determinacy, and completeness as well as to further raise the code level abstraction that should enable more concise and natural implementations of adaptive elements.

The resulting architecture model is used as an input into model manipulation tools such as model verifiers or source code generators. The next chapter presents details about these tools, which have been incorporated into a software package called the *ACTRESS modeling environment*.

CHAPTER 5

The Actress Modeling Environment

In the last two chapters we have presented a domain-specific modeling language for feedback control architectures. We discussed the use of models offering higher expressiveness, ease of use and their potential to lower accidental complexities that in turn should result in a higher productivity. For this to be true, however, models have to be associated with software development tools that automate tasks such as model construction and code generation [Sendall and Kozaczynski, 2003]. In this chapter we introduce a set of tools facilitating FCDL development that has been implemented inside a modeling environment called Actress.

The aim of Actress is to provide support for an integrated development of external self-adaptive software systems using FCDL, *i.e.*, integrating self-adaptive control mechanisms into software systems through feedback control architectures. We do not focus on developing the control mechanisms themselves. For this, there already exist sophisticated tools [Hellerstein et al., 2004] such as MATLAB or Ptolemy (*cf.* Section 2.2.2 on page 29).

In its core, Actress consists of a series of model transformations and verification processes automatizing various aspects of FCDL development. We start by presenting the modeling support for authoring FCDL models. Next, we give an overview of the code generation process that synthesizes system implementation. This is followed by the discussion of the FCDL model verification support. At the end of the chapter, we discuss Actress integration into the Eclipse *Integrated Development Environment* (IDE)[1].

5.1 Modeling Support

The Actress modeling support provides a reference implementation of the FCDL metamodel and tools facilitating FCDL models authoring. The implementation is based on

[1] http://www.eclipse.org

the EMF meta-modeling technology. Figure 5.1 shows a high-level overview of the model support components and artifacts. The heart of the modeling support is a domain-specific

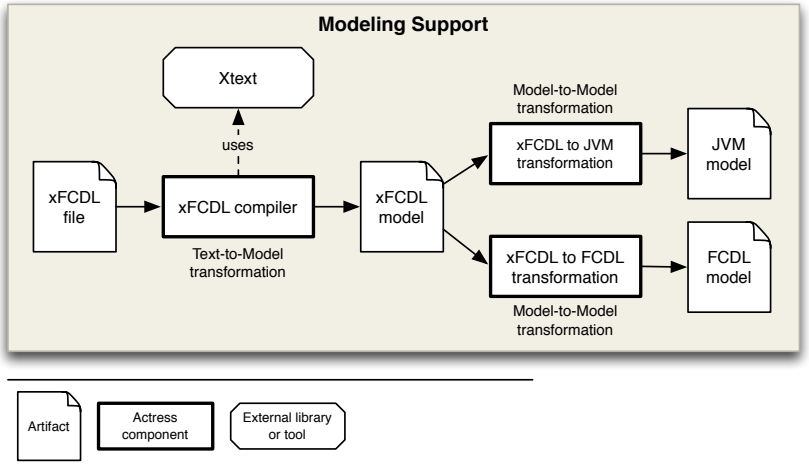


Figure 5.1: Overview of the Actress modeling support

language called *Extended Feedback Control Definition Language* (xFCDL) for creating FCDL models. It is a textual DSL for creating FCDL models also supporting modularization and adaptive element implementation using a Java-like expression language. xFCDL is built using Xtext[2], which is an EMF-based framework for domain-specific and general-purpose programming languages development. It covers many aspects of a language infrastructure including sophisticated Eclipse IDE integration (*cf.* Section 5.4).

In the next subsection we explain why we have created a DSL for FCDL modeling. Next, we give an overview of the main concepts and features of the DSL and how it can be used to specify adaptive element implementations. This is followed by details about the expression language integration and how it transforms into FCDL.

Why Eclipse and Eclipse Modeling Framework? We have chosen to use the Eclipse platform and EMF for Actress implementation as they are both open source and mature technologies used extensively in both academia and industry. EMF comes with a wide range of supporting tools and technologies that facilitate domain-specific modeling[3]. Furthermore, EMF is well integrated within the Eclipse IDE, providing a complete integrated development experience.

There are other meta-modeling environments such as MetaCase's MetaEdit+[4] or Jetbrain's MPS [5]. However, they are either commercial (MetaEdit+) or integrated with a commercial IDE (MPS).

[2] http://www.eclipse.org/Xtext/
[3] http://eclipse.org/modeling/
[4] http://www.metacase.com/mwb/
[5] http://www.jetbrains.com/mps/

5.1.1 Why a Domain-specific Language?

There are at least tree possible ways how to create FCDL models. One approach is to use directly the FCDL meta-model and create its instances using some generic modeling editor offered by the meta-modeling technology or directly by the provided API. The other two approaches involve creating a new FCDL concrete syntax or a completely new domain-specific language with transformation to FCDL. This subsection discusses these options and motivates our choice.

Direct Meta-Modeling using an Editor or API Most of the meta-modeling environments provide some tool support and API for creating models. EMF comes with a simple generic tree-based editor capable of generating a more sophisticated and customizable version for a concrete meta-model. In our case, the two main problems of using either of these options is that the tree visualization is not appropriate for FCDL and directly working with the FCDL meta-models is not very convenient (*cf.* Figure 3.10 on page 56). To use the EMF meta-modeling API is even less desirable since the level of abstraction at which the models are described is the one of the programming language, *i.e.*, Java.

Graphical or Textual Concrete Syntax Among different concrete syntaxes that exist, graphical and textual are the two most common forms. In Chapter 3, an informal graphical notation for FCDL was introduced. However, it is not formal enough for a complete description (*e.g.* does not include information about data types). While it could be extended to cover the full FCDL specification, doing so only makes sense if it is accompanied by a tool support to be used to create FCDL models. However, the effort of developing a usable graphical editor is usually rather high[6] in comparison to the effort of creating a text editor.

A textual concrete syntax is defined by a grammar and it involves a compiler that translates the text into the target meta-model instances. In MDE, this process is referred to as *Text-to-Model Transformation* (T2M) and there exist several tools that can automatize it. Examples of these tools in EMF include Xtext[7], EMFText[8], MontiCore [Krahn et al., 2010] and TCS [Jouault et al., 2006]. They generate complete compilers including scoping and linking[9] support based on language grammars and meta-model definitions. In addition, they are usually well integrated within the Eclipse IDE and provide rich editing support including syntax highlighting, content assists, quick fixes and other features known from modern development environments.

In our case, the problem with the T2M approach is that the grammar parser rules defining the mapping between the textual tokens and the modeling elements have to match the target meta-model elements. This matching is usually defined using EBNF-like (*Extended*

[6] This of cause greatly depends on the meta-modeling environment. In case of Eclipse GEF (`http://www.eclipse.org/gef/`), which is standard framework for EMF based graphical editors, a simple editor of only three elements already consists of 65 files with nearly 5,000 lines of code [Kelly, 2004].

[7] `http://www.eclipse.org/Xtext/`

[8] `http://www.emftext.org/index.php/EMFText`

[9] They allow to specify cross-linking information directly in the grammar and thus next to parsers and lexers can also generates additional facilities for linking model element references together.

Backus-Naur Form) expressions [ISO14977, 1996] with one symbol for each concept from the abstract syntax. Therefore, the concrete syntax would be tied to the FCDL meta-model and developers will have to define all the FCDL elements, just like in the case of EMF editors.

Domain-Specific Language The above mentioned limitation only appears when the concrete syntax has to map to an existing meta-model that is fixed and cannot (or should not) be changed. To go around this issue, we define a new language with its own meta-model that is separated from FCDL. While this involves additional effort, especially since we also have to provide transformations between the two meta-models, it brings the maximum flexibility for the grammar definition. Moreover, this separation allows for creating new concepts that are not and should not be part of the original meta-model. For example, let us consider specifying behavior of an adaptive element, *i.e.*, associating a concrete implementation (*e.g.* a Java code) to an interaction contract (Section 4.2). The FCDL is a technologically-agnostic modeling language and as such, it is not primarily concerned with implementation details beyond the feedback control loop architectures. Additional details such as the name of a Java class that implements a particular adaptive element behavior can only be added using annotations (Section 3.3.8). With the new language, however, we can provide support for the implementation specification directly in the language. While this will make the language tied to a particular set of tools (*e.g.* the code generator), it offers substantial gains in productivity.

5.1.2 xFCDL in a Nutshell: Modeling FCL Architectures

This and the next subsections present the main xFCDL constructs. Details about the language abstract and concrete syntaxes are provided in Appendix C.1 and C.2 respectively. For better illustration, the language features are demonstrated using an excerpt of the `QOSControl` taken from the running example shown in Figure 5.2.

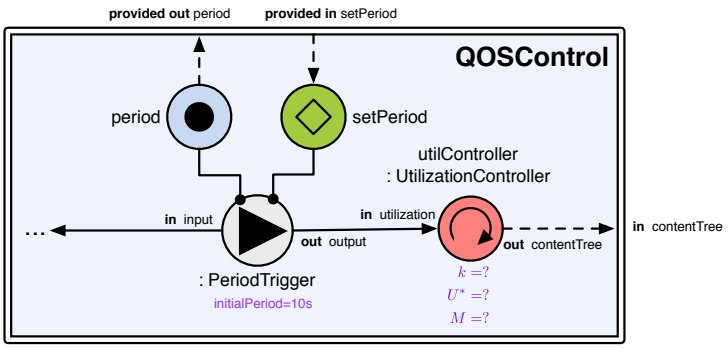


Figure 5.2: An excerpt of the `QOSControl` composite used for the xFCDL illustration

Overview xFCDL is a domain-specific modeling language for defining complete FCDL models including adaptive element implementations in a convenient and expressive tex-

tual form. Essentially, an xFCDL file consists of one or more adaptive elements or composite type definitions. The language is close to Java and it uses some of the Java concepts such as modularization and type system as well as naming convention [Sun, 1997]. This should help developers familiar with Java to quickly understand the notation. Moreover, there is a close interoperability with Java[10] for the adaptive element behavior specification that will be detailed later in sections 5.1.3 and 5.1.4.

Organization FCDL is not concerned with modularization and cross-model element references[11]. All adaptive element types are contained under the same control system model instance. xFCDL introduces a concept of modularization whereby feedback control architectures can be organized in multiple files. It uses the Java package mechanism to organize adaptive elements into different namespaces. The import mechanism also mimics the one of Java. For example, following declarations

```
package demo.webserver

import fcdl.math.*
import fcdl.sys.NewSystemProcessNotifier
```

allow to directly use any adaptive element within the `demo.webserver` and `fcdl.math` packages as well as `NewSystemProcessNotifier`. Any other adaptive element will have to be referred to using its fully qualified name.

Data Types Besides importing existing adaptive elements, an import statement can also be used to import just about any other Java class. These classes can then be referred to in the data type definitions (Section 3.3.2). The xFCDL follows the same type conformance rules as Java. With an implicit import of `java.lang.*`, it makes all the standard Java data types available straight in the xFCDL. The language also differentiate class types from primitive types (*e.g.* `int` and `java.lang.Integer`). The xFCDL compiler is responsible for creating the necessary concrete data type instances in the final FCDL model.

The xFCDL language does not provide any constructs for defining new data types. This means that all data types must be imported and thus if a new data type is needed it has to be defined *a priori* in Java[12]. The main reason is that adding a straight-forward data type definition support that is interoperable with Java will significantly increase the language size. It will have to include constructs for extending existing Java types, implementing interfaces, access modifiers and defining and overriding methods eventually turning xFCDL into a GPL. However, having a lightweight data type system definition that allows to only construct a simple record-like structures or physical units might be eventually beneficial and it is one of the subject of our further work (*cf.* Section 9.2.1).

Adaptive Element Types The architecture description consists primarily in adaptive element types definitions. The following code shows an example of how to create the `Pe-`

[10] More precisely with any Java Virtual Machine language that compiles to Java classes.

[11] The EMF implementation of FCDL actually supports cross-model referencing because it inherits this mechanism from EMF, however, there is no first-class support for it directly in the meta-model itself.

[12] Or any other language that compiles to Java classes

riodicTrigger processor from the running scenario. The full xFCDL definition of the scenario is listed in appendix D.1 (line 94-143).

```
1  package demo.webserver
2
3  active processor PeriodicTrigger<T> {
4    push in port output: T
5    pull in port input: T
6    self port selfport: long
7
8    provided sensor period: Duration
9    provided effector setPeriod: Duration
10
11   property initialPeriod: Duration = 10.seconds
12
13   // ...
14 }
```

Next to a package declaration, this listing defines a new adaptive element type, an active processor called `PeriodicTrigger` (line 3). At the same time it also defines a data type parameter `T`. The symbol `T` can be then used inside the adaptive element definition for any data type specification. It is used on lines 4 and 5 that respectively declare the push output and pull input ports. This makes it a *polymorphic adaptive element* since it can be used with varying data types. There can be any number of data type parameters (separated by a comma). Furthermore, just like Java, xFCDL also supports type bounds using the **`extends`** keyword.

Because, it is an active processor, it implicitly defines a self port (`selfport`). We only need to redefine it (line 6) if we need to: change the default name (`selfport`)[13] or specify its data type which is `void` by default. Lines 8 and 9 define reflection capabilities of this adaptive element. That includes a provided sensor `period` and a provided effector `setPeriod`, both using `Duration` data type (a finite time duration defined in the `fcdl.lang` package that is implicitly imported). Finally, on line 11, a required property `initialPeriod` of the same data type is declared together with a default value of 10 seconds.

The next example shows the xFCDL code for the `UtilizationController` controller, defining two ports, three properties and no data type parameters.

```
1  controller UtilizationController {
2    in push port utilization: double
3    out push port contentTree: double
4
5    property k: double
6    property targetUtilization: double // U^*
7    property M: int
8
9    //...
10 }
```

Intuitively, other two adaptive element roles are created by using corresponding keywords: **`sensor`** and **`effector`**.

[13] The reason why it called `selfport` and not just `self` as it was defined in Definition 5 is that `self` is internally used by Xbase to refer to containing scope of lambda expressions.

Composition Section 3.3.4 presented the FCDL support for composing multiple related adaptive elements together in order to form higher-level structures. The following listing shows an example of a composite definition in xFCDL. Concretely, it shows an excerpt of the `QOSControl` composite definition from Figure 5.2 featuring the two above defined adaptive element types.

```
1  composite QOSControl {
2   // composite ports
3   out push port contentTree: double
4
5   // composite provided ports
6   provided sensor period: Duration
7   provided effector setPeriod: Duration
8
9   // composite properties
10  property k: double
11  property targetUtilization: double
12  property M: int
13  // ...
14
15  feature scheduler = new PeriodicTrigger<Double> {
16   // redefines the default value of initialPeriod
17   initialPeriod = 30.seconds
18  }
19
20  feature utilController = new UtilizationController {
21   // references the composite properties
22   k = this.k
23   targetUtilization = this.targetUtilization
24   M = this.M
25  }
26  // ...
27
28  // create a communication channel
29  connect scheduler.output to utilController.utilization
30  // ...
31
32  // port promotions
33  promote scheduler.period
34  promote scheduler.setPeriod
35  promote utilController.contentTree
36  // ...
37 }
```

Since composites are also adaptive elements, they can also declare ports (line 3), provided sensors and effectors (line 6 and 7) and properties (lines 10-12). Additionally to adaptive elements, composites define an assembly of other adaptive elements. On line 15 a new instance of `PeriodicTrigger` named `scheduler` is created, including a data type argument specification for the `PeriodicTrigger` data type parameter `T`.

Instead of specifying concrete data types, we could make the composite polymorphic by adding a data type parameter that would now serve as the data type argument for the instantiation. For example

```
composite QOSControl<T> {
 // ...
 feature scheduler = new PeriodicTrigger<T> { /* ... */ }
```

```
  // ...
}
```

The body of an instantiation (the code between the curly braces) gives an opportunity to specify values of the element properties. Similarly, on line 20, a new instance `utilController` of the `UtilizationController` is created. For the property specification, an implicit **`this`** reference is used to refer to the properties defined in the surrounding composite. Finally, line 29 connects the `scheduler` output port to the input of the `utilController` and lines 33-35 promote the remaining ports to the corresponding ports defined in the composite.

Distribution The **`new`** operator creates a new instance, a new contained feature. In the case of distributed adaptive element, we might instead need to only reference a remotely deployed element (*cf.* Section 3.3.6). In xFCDL, it is supported by the **`ref`** keyword.

```
1  // deployed at host remote-apache
2  composite Apache {
3    feature apache = new ApacheWebServer {
4      // ...
5    }
6    feature control = ref ApacheQOS.control @ "akka://remote-main/user/ApacheQOS/control"
7  }
8
9  // deployed at host remote-main
10 composite ApacheQOS {
11   feature control = new QOSControl {
12     // ...
13   }
14   feature apache = ref Apache.apache @ "akka://remote-apache/user/Apache/apache"
15   // ...
16 }
```

In this example, we have two remotely deployed composites that reference features from one another (*cf.* Figure 3.14), lines 6 and 14. The URI indicating the remote features locations are implementation specific. In the listing above we use the Akka URIs since the Actress domain framework is based on Akka (*cf.* Section 5.2.2).

Interaction Contracts The remaining part of the adaptive element structural definition is the specification of its interaction contract(s). This is done by including one or more **`act`** keyword(s) that define basic interaction contract(s).

```
1  active processor PeriodicTrigger<T> {
2    push in port output: T
3    pull in port input: T
4    self port selfport: long
5
6    provided sensor period: Duration
7    provided effector setPeriod: Duration
8    // ...
9
10   act activate(selfport; input; output?)
11   act onSetPeriod(setPeriod; ; period?)
12 }
```

The corresponding interaction contract definition is between lines 10 and 11 declares two interaction contracts:

$$\begin{aligned} \texttt{activate} \text{ (line 10)} &= \langle\, self; \Downarrow (\texttt{input}); \Uparrow (\texttt{output?}) \,\rangle \\ \texttt{onSetPeriod} \text{ (line 11)} &= \langle\, \Uparrow (\texttt{setPeriod}); \emptyset; \Uparrow (\texttt{period?}) \,\rangle \end{aligned}$$

The definition in xFCDL has the following structure: **act** $name$ $(A, R, E)[?]$. After the **act** keyword, the $name$ assigns a name to the interaction contract that will be used for the activation method name and the A, R, E define respectively the ports involved in the activation, data requirements and data emission parts of the contract. Appending `?'' after an emission port makes the emission optional. Similarly, `?' following the interaction contract specification denotes an optional contract. The ports are separated by commas and in the case of input disjunction for the activation condition by lowercase letter `v' ($\texttt{input}_1$ `v ... v` $\texttt{input}_n$).

Annotations Similarly to FCDL, xFCDL supports annotations to include additional details to modeling elements. The annotation are defined in Java-like syntax and the arguments values are always string literals. For example, marking `ApacheQOS` composite as main is expressed using the `@Main` annotations as:

```
@Main(name="actress")
composite ApacheQOS { /* ... */ }
```

Annotations can be attached to all xFCDL declarations (to an adaptive element, a port, a property, an interaction contract, a connection, a promotion and a feature).

5.1.3 xFCDL in a Nutshell: Adaptive Element Implementation

The previous subsection has reviewed the xFCDL concepts related to the structural description of the elements composing feedback control loop architectures. From such a description it is already possible to synthesize a partial implementation of the adaptive elements. What is missing, is the adaptive elements behavior, *i.e.*, the implementation of the interaction contracts activation methods. There is no restriction to what an adaptive element behavior should be. For example, the elements from the running scenario (*cf.* Section 3.2 and 3.3.1) use mostly arithmetic expressions, but the web server touchpoints also use IO and other libraries such as regular expression for the log file parsing.

One way to specify a behavior is thus to directly modify or extend the generated code in a GPL like Java. However, this comes at the price of having a closely related information in two different places maintained at two different levels of abstraction in two different languages. Therefore, in order to provide a more integrated approach, we extend the xFCDL language with expressions that can be used to specify adaptive element behavior directly in the xFCDL files.

The support is based on an expression language called Xbase[14] that conceptually and syntactically resembles the Java language. It is a statically typed expression language that

[14] http://www.eclipse.org/Xtext/documentation.html#xbaseLanguageRef_Introduction

additionally includes support for some more advanced concepts such as lambda expressions, type inference and operator overloading. Developed as a part of the Xtext framework, it can easily be mixed with other Xtext languages.

Xbase expressions are particularly convenient for implementing adaptive elements that are based on mathematical equations. For example, the `UtilizationController` is responsible for computing severity of adaptations, G, using equation (3.2):

$$\begin{aligned} E = &\quad U^* - U \\ G = &\quad G + kE \end{aligned}$$

where U^* and U are respectively the target and the current system utilization (*cf.* Section 3.2). The Xbase implementation allows one to directly express this equation in the xFCDL:

```
controller UtilizationController {
 // ...

 // beginning of Xbase implementation
 implementation xbase {
  var G = M

  // implementation of the
  //  act activate(utilization;;contentTree)
  // interaction contract
  act activate {
     // computes the error
     val E = targetUtilization - utilization

     // computes new extend of adaptation
     G = G + k * E

     // correct bounds
     if (G < 0) G = 0
     if (G > M) G = M

     // returns the result
     G
  }
}
```

The code in the Xbase implementation can use any Java library. It is thus possible to implement more complex formulas and algorithms leveraging Java libraries for linear algebra [15], statistics[16] and the like. It can also be used for technical elements such as `PeriodicTrigger` whose implementation is shown below in Listing 5.1.

```
   implementation xbase {

      // variable definitions
      var currentPeriod = initialPeriod // reference a property
      var task: Cancellable

      // life-cycle callbacks
      def init {
```

[15] http://la4j.org/
[16] http://commons.apache.org/proper/commons-math/

```
9           // start the timer task
10          reschedule
11      }
12
13      def destroy {
14          // cancel the timer task
15          task.cancel
16      }
17
18      // interaction contract activation method implementations
19      act activate {
20          log.info("Activate at "+selfport.get)
21
22          // get data from the input port
23          val data = input.get
24          // check for nulls and eventually output
25          if (data != null) output.put(data)
26          else log.info("No data available on the input port")
27      }
28
29      act onSetPeriod {
30          // checks if the setPeriod - a value pushed through the provided
31          // sensor is different than the current one
32          if (setPeriod != currentPeriod) {
33              // if so - update, reschedule and notify through provided sensor
34              currentPeriod = setPeriod
35              reschedule()
36              period.put(currentPeriod)
37          }
38      }
39
40      // auxiliary methods
41      def reschedule {
42          // cancel if it has been already scheduled
43          if (task != null && !task.isCancelled) task.cancel
44
45          // scheduler is one of the service provided by the domain framework
46          // we schedule a closure with 2 seconds delay and currentPeriod as period
47          task = context.scheduler.schedule(2.seconds, currentPeriod) [|
48              // closures are are denoted by square brackets
49              // once executed it pushes current time to the selfport
50              // selfport is the only port, that is available in this scope
51              selfport.put(System::currentTimeMillis)
52          ]
53      }
54  }
55 }
```

Listing 5.1: Xbase implementation of `PeriodicTrigger`

The Xbase block starts at line 12 indicated by the **`implementation xbase`** keywords. It consists of four parts:

- declarations and definitions of any number of variables (lines 4 and 5),
- life-cycle callbacks for element's initialization and destruction (lines 8-16),
- interaction contract implementations (lines 19-38), and
- any number of auxiliary methods (lines 41-54).

During the code generation, Xbase code is compiled into regular Java code (*cf.* Section 5.2.1). For example Listings D.2 and D.3 in the appendix show the complete Java code inferred from the above `PeriodicTrigger` definition.

Xbase provides a convenient way of specifying adaptive elements implementation directly in xFCDL, however, it might not always be the most suitable option. In the next section we show how a custom implementation based on Java or Scala can be used instead.

Higher-Order Adaptive Elements There is another advantage of using Xbase as it enables creating *higher-order adaptive elements* by using lambda properties. For example, Section 3.3.5 showed an FCDL schema of dynamic deployment of the running example (*cf.* Figure 3.12). It involved a sensor broadcasting a notification every time a new system process has been started by an operating system. This notification was piped into an `ApacheProcessFilter` to filter out non-Apache processes. Let us now imagine that we want to reuse this FCDL model with a different web server, *e.g.*, Lighttpd. We could reuse all the adaptive elements, but we would have to create a new process filter. Instead of the `ApacheProcessFilter` we would have to implement a `LighttpdProcessFilter` that essentially just replicates the logic of the `ApacheProcessFilter` but use *lighttpd* instead of *httpd* in the filter predicate. Using Xbase support for higher-order functions, we can abstract the system process filtering by defining a property that acts as a predicate, *i.e.*, a function that takes a process information and returns either *true* or *false* depending on whether the process should be pushed further or skipped. Concretely, in xFCDL it can be achieved as follows:

```
processor ProcessFilter {
  // ...
  // one parameter predicate
  // it takes a ProcInfo as a parameter and returns a boolean
  property filter: (ProcInfo)=>boolean
  // ...
}

composite ApacheQOSDeployer {
  // ...
  feature apacheFilter = new ProcessFilter {
    // define the filter predicate for Apache
    filter = [proc | proc.name == "httpd"]
  }
}

composite LighttpdQOSDeployer {
  // ...
  feature lighttpdFilter = new ProcessFilter {
    // define the filter predicate for Lighttpd
    filter = [proc | proc.name == "lighttpd"]
  }
}
```

Furthermore, we can use the data type parameters to create a polymorphic one parameter predicate processor:

```
processor ItemFilter<T> {
  // ...
  // one parameter polymorphic predicate
  property filter: (T)=>boolean
  // ...
}

composite ApacheQOSDeployer {
  // ...
  feature apacheFilter = new ItemFilter<ProcInfo> {
    // define the filter predicate for Apache
    filter = [proc | proc.name == "httpd"]
  }
}
```

This support allows to define more generic adaptive elements further increasing possibility of their reuse.

5.1.4 xFCDL to JVM Model Transformation

Xbase is a language library that can be embedded into other Xtext languages. In order for this embedding to work, domain-specific concepts of the host language have to be translated into Xbase concepts, *i.e.*, JVM model elements such as Java classes, fields and methods. By providing this mapping Xbase can take responsibility of proper expressions scoping, method invocation resolutions and type conformance checking. Essentially, this translation is a model-to-model transformation between xFCDL abstract syntax and the Xbase JVM model. Concretely, the transformation involves creating an adaptive element delegate and adaptive element act classes as they were described in Sections 4.1.1 and 4.2.8 respectively. At this point we are only interested in the class structure so element members (*e.g.*, ports and properties) can be referenced and type checked. However, this transformation serves as a basis for the code generator that additionally synthesizes the implementation of the inferred methods (*cf.* Section 5.2.1).

Following is a high level overview of the JVM inference. This mapping is based on the semantics defined in the previous chapter (Section 4.1 and 4.2) using the types presented in Figure 4.1 and 4.13.

Adaptive Element Delegate Inference An adaptive element delegate is a Java class that extends from the abstract `AdaptiveElementDelegate` class (*cf.* Figure 4.1). The inference rules include:

– data type parameter map into the Java class type parameters,
– properties map into instance variables and constructor parameters,
– ports map into instance variables,

Adaptive Element Acts Inference An adaptive element act is a Java class whose inference rules include:

– data type parameters map into the Java class type parameters,
– properties map into instance variables and constructor parameters,

– Xbase variables map into instance variables,
– `selfport` maps into an instance variable,
– life-cycle methods, interaction contract acts and any other auxiliary methods map into instance methods with appropriate return type and parameter list.

Example Figure 5.3 shows a concrete example of the JVM model inference for the `PeriodicTrigger`. The actual code that is synthesized by the code generator is listed in Appendix D.2 for the adaptive element delegate and in Appendix D.3 for the adaptive element act.

```
active processor PeriodicTrigger<T> {
  push in port output: T
  pull in port input: T
  port selfport: long

  provided sensor period: Duration
  provided effector setPeriod: Duration

  property initialPeriod: Duration

  act activate(selfport; input; output?)
  act onSetPeriod(setPeriod; ; period?)

  implementation xbase {
    var currentPeriod = initialPeriod
    var task: Cancellable

    def init {
      // ...
    }
    def destroy {
      // ...
    }

    act activate {
      // ...
    }
    act onSetPeriod {
      // ...
    }

    def reschedule {
      // ...
    }
  }
}
```

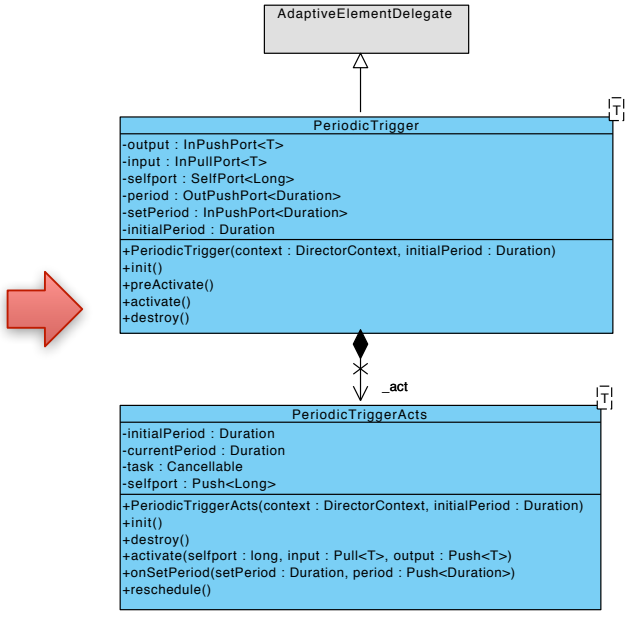


Figure 5.3: xFCDL to JVM model transformation

5.1.5 xFCDL to FCDL Transformation

The final missing part is the transformation that maps xFCDL models into FCDL models (FCDL types package). By looking at both meta-models (xFCDL in Appendix B.2 and FCDL in Appendix C.1), the transformation is rather intuitive and the concrete list of rules is given in Appendix C.3. The main issue in the relation between xFCDL and FCDL model elements are the data types.

Data Types A data type in FCDL model is used for two purposes: type conformance checking and code generation. In xFCDL a data type is represented using an Xbase `JvmTypeReference` element which references some Java type. Additionally, Xbase provides a service that checks type conformance of two `JvmTypeReferences`. Using this service significantly simplifies the implementation of the data type conformance verifier (*cf.* Section 3.3.2). Therefore, while converting the `JvmTypeReference` into `ConcreteDataType`,

which is used for data type representation in FCDL, we use the EMF adapters (*cf.* [Steinberg et al., 2008, Section 16.2]) to store the original reference to `JvmTypeReference` so it can be used later by the type checker. For the *name* property of the `ConcreteDataType` a fully qualified Java class name of the original `JvmTypeReference` is used. The very same applies for `JvmTypeParameter` that is used to represent data type parameter in xFCDL.

5.2 Code Generation Support

The Actress modeling environment provides a code generation support that synthesizes system implementation from FCDL models. Figure 5.4 shows a high level overview of its components and artifacts.

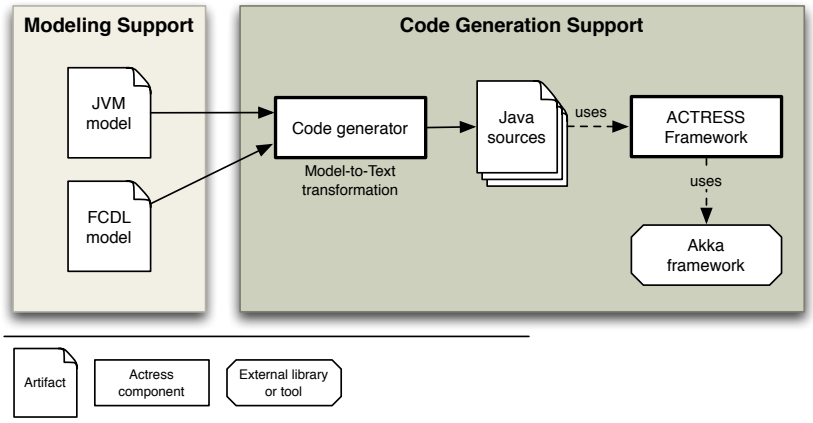


Figure 5.4: Overview of the Actress code generation support

Essentially, it consists of model-to-text transformations that take an FCDL model together with an inferred JVM model (*cf.* Section 5.1.4) and output Java source files. The transformation is model-aware[17] [Fowler, 2010] and the generated code is using the Actress domain framework. The framework provides necessary abstractions for seamless mapping of FCDL adaptive elements and composites into lower-level actors. It also constitutes of a runtime environment for executing these actors, essentially implementing the semantic rules described in Chapter 4. Code generation for a particular framework makes the code generator tightly coupled with it. On the other hand, the more abstractions the framework provides the less work is done in the code generation and the resulting code is more readable and easier to test and debug. Furthermore, it only interfaces with the framework API and thus multiple implementations can be used. It is important to stress here that while this section illustrates the use of a particular technological stack for the implementation, *i.e.*, Java and Akka[18], it presents just one possibility. For example, we could equally use C++ and libcppa[19] as FCDL is a technologically agnostic model. This is further discussed in Section 8.2.1 where an alternative FCDL implementation is presented.

[17] Generated code has an explicit simulacrum of the model semantics usually based on a domain framework.
[18] `http://akka.io`
[19] `https://github.com/Neverlord/libcppa`

We begin this section presenting the code generator itself that will be followed by an overview of the Actress framework.

5.2.1 Code Generator

Code generator translates FCDL model elements into source code artifacts that together form the self-adaptive layer that controls the target system. It consists of an actor runtime and a network of hierarchically composed actors. The top actor in this hierarchy represents a main FCDL composite. For example, Figure 5.5 shows the actor hierarchy representing the adaptive elements from the running scenario (*cf.* Figure 3.8) that will be deployed in the Actress runtime.

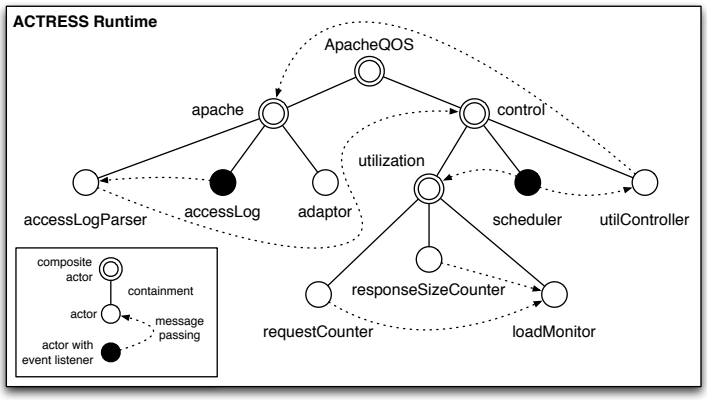


Figure 5.5: The hierarchy of the running example actors in the Actress runtime

Because the FCDL input model is already an actor-oriented model, the source code transformation is rather straightforward as it does not need to build any other intermediate representation. Taking FCDL instances and the inferred JVM model, the following artifacts are generated:

- For each non-composite adaptive element an *adaptive element delegate* class is generated.
- For each non-composite adaptive element either a complete or a ``*skeleton*'' implementation of *adaptive element act* class is generated based on whether Xbase was used.
- For each composite a *composite delegate* class is generated.
- For each main composite a *composite launcher* class is generated.

Following the generation gap pattern, the sources of xFCDL projects are split in two directories `src-gen` and `src`[20].

The code generator emits Java code. There are two reasons for that. First, the code generator relies on the Xbase compiler to transform Xbase expressions into code and currently

[20] A concrete example of what is being generated for the running scenario is shown in Figure 5.10 on page 121.

it only supports Java. Second, despite the fact that we use Scala for Actress implementation and for code illustrations in this thesis, by generating Java code we potentially attract a wider audience. Nevertheless, the domain framework has both Java and Scala API and adaptive element skeletons can be implemented in either languages.

In the rest of this subsection we discuss the different generated artifacts.

Adaptive Element Delegate and Act In Section 5.1.4 we have described the process of inferring the JVM model from xFCDL elements. The exact same process is also used for source code generation. The code generator creates the very same classes and additionally synthesizes their implementations. Namely, for adaptive element delegates it emits code for life-cycle methods (*i.e.*, `init`, `destroy`, `preActivate` and `activate`) following the semantic rules defined in Chapter 4, mainly in Section 4.1.5 and 4.2.8. A concrete example of what is synthesized for the `PeriodicTrigger` delegate is shown in Listing D.2.

Essentially, the `init` and `destroy` methods are simply delegated to the adaptive element act class. The `preActivate` method represents a logical disjunction of activation conditions of the adaptive element interaction contracts. The `activate` method follows the contracts delegating to the appropriate methods defined in the corresponding adaptive element act class. This behavior is shown graphically below in Figure 5.6.

An implementation of adaptive elements act includes the `init` and `destroy` life-cycle methods and the activation methods corresponding to interaction contracts. There are two possibilities. Either, they were implemented using Xbase and thus the Xbase compiler generates the corresponding Java code, or no implementation was given in which case an empty skeleton is generated. An example of the code generated from the Xbase implementation of the `PeriodicTrigger` is shown in Listing D.3.

Custom Implementation With Xbase expressions the code generator can synthesize a complete system. However, Xbase might not always be the most suitable language. For some of the more complex touchpoints that need to interface with third-party libraries, a direct implementation in Java or in Scala might be preferred[21]. The generated skeleton class is therefore provided for a custom implementation using a GPL. As we have pointed above, in order not to mix generated and user modified code the custom implementation is separated from the generated one by inheritance and resides in a separate directory. We use `ActCustom` suffix for the user-modifiable class.

Composites Delegate and Actor A composite is responsible for managing the features it contains, *i.e.*, for the features life-cycle, message forwarding over promoted ports and port connections. It does not provide any particular behavior on its own and thus there is no need to generate any adaptive element act class. Moreover, since all composites behave the same, most of the implementation can be abstracted into a base superclass in the domain framework (*cf.* Section 5.2.2 on page 114). The code generator therefore

[21] Xbase does not yet (as of version 2.4) support nested classes and implementation of interfaces with more than one method

only needs to generate a code related to composite initialization where it iterates over the composite structure to create contained features, look up remote feature references and sets up their port connections and promotions. An example of a composite delegate for `UtilizationMonitor` is shown in Listing D.5.

Composite Launcher The Actress code generator also generates support for launching FCDL applications. For each main composite, the code generator emits a Java class with the `main` method that initializes the Actress runtime and deploys the composite into it. Such a launcher can be further modified by developer to properly configure the composite and the actor runtime. An example of the generated launcher for `ApacheQOS` composite is shown in Listing D.4.

5.2.2 The Actress Framework

The Actress domain framework is an implementation of the FCDL model of computation (*cf.* Section 4.1) on the top of Akka. Concretely it provides an implementation of the adaptive element director (*cf.* Section 4.1.1), the composite director and the Akka-based runtime. The framework is implemented in Scala, but it also includes a Java API that is used by the generated code.

Why Akka? Akka is a framework and a runtime for highly concurrent, distributed, and fault tolerant actor-based applications on the JVM. In its core it supports location transparency, and it provides two important message delivery guarantees: at-most-once delivery, and message ordering per sender–receiver pair. It is designed to be scalable and lightweight (*cf.* Section 8.2.3) and it comes with both Java and Scala APIs.

Adaptive Element Director The adaptive element director is an Akka actor that is responsible for handling port connections, message communications and also for the associated delegate life-cycle and interactions. All non-composite adaptive elements are using the same director implementation.

Figure 5.6 shows a concrete example of an adaptive element director execution associated to `PeriodicTrigger`. It follows the generic schema shown in the previous chapter (*cf.* Figure 4.2), only this time it depicts a concrete implementation. There are 6 executions. The first two are related to the port connections, informing the director about the targets of its ports using the `CONN` message. Next, an `INIT` message is sent to initialize the actor. The `CONN` and `INIT` messages are sent from the composite that owns the instance of this `PeriodicTrigger`. In this particular example it is the `QOSControl` composite (*cf.* Figure 3.8). The executions 4 and 5 are examples of `PUSH` messages over the self port and over the provided effector. The last message, `KILL`, is sent by the Actress runtime when it is shutting down allowing the actor to gracefully terminate.

Composite Director Similarly to the adaptive element director, the composite director is an Akka actor that is responsible for handling FCDL composites, *i.e.*, managing life-

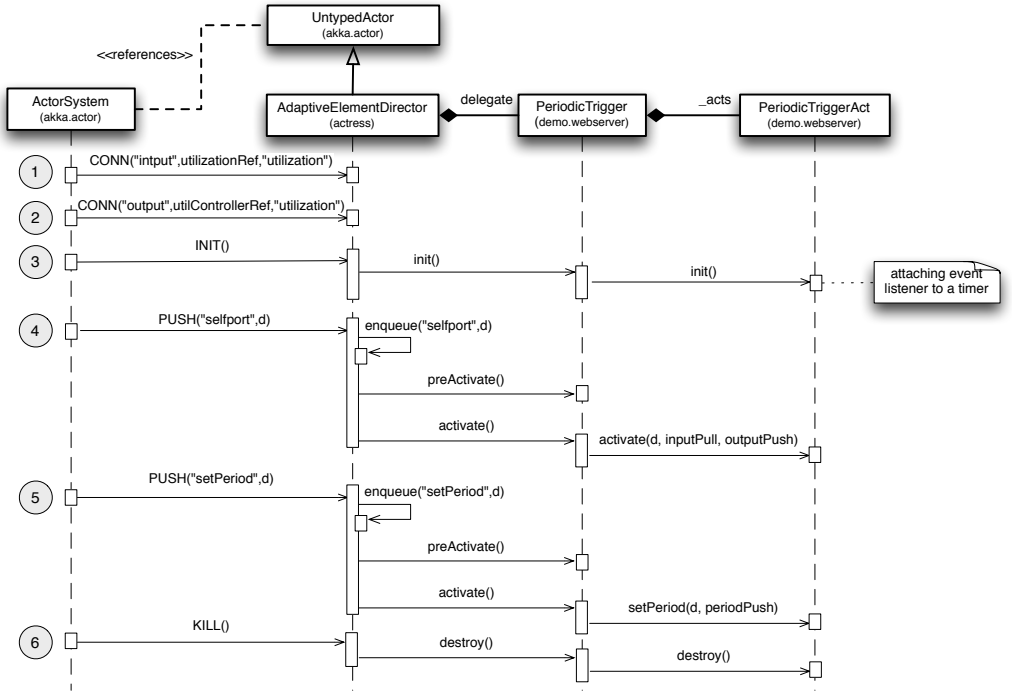


Figure 5.6: Example of an adaptive element director execution

cycle of contained features (initialization and termination) and message forwarding over promoted ports. Similarly to the case of adaptive elements, all composites are also using the same director implementation that associates a concrete composite delegate which is automatically synthesized by the code generator. A composite delegate is a specialized version of the adaptive element delegate. As we have noted above, all composites have the same behavior that only varies depending on the composite structure. Therefore, out of the four life-cycle methods only the `init` method has to be generated. This is where the composite structure gets exposed and features registered. The implementation of the other methods is provided by the composite delegate super class. The `init` method is further split into a number of dedicated methods that are responsible for the different composite initialization tasks that involves following steps: (1.) creation of all contained features, (2.) lookup of all referenced features, (3.) connection of features ports, (4.) setup of promoted ports, and (5.) initialization of all contained features.
Because of the dynamic nature of the actor system we use that particular initialization order to make sure that ports are connected before the adaptive elements are initialized. This is to prevent active elements pushing or pulling data from disconnected ports.

Composite life-cycle consists of four states (*cf.* Figure 5.7):

– *Started.* The initial state of a newly started composite. In this state it answers to port connection messages and waits for the `INIT` message which will move it to the next state.

– *Initializing.* During initialization it consecutively executes the five initialization tasks

from the composite delegate. It waits for the all the featured adaptive elements to be started or looked up and then it moves to the running state.

– *Running.* When running, the composite responds to push and pull requests, forwarding the messages to the appropriate endpoints of the promoted port connections. It can also rewire ports and answer a `KILL` message by moving to the *terminating phase*.

– *Terminating.* The termination process simply sends `KILL` messages to all the contained features.

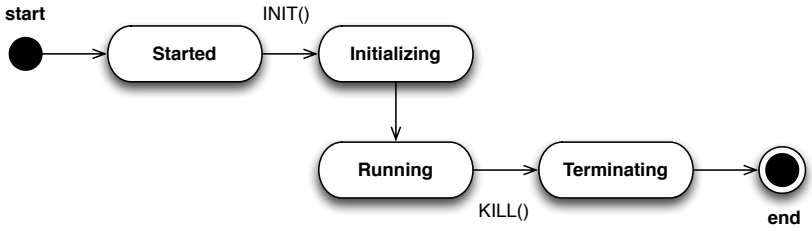


Figure 5.7: Composite director life-cycle

The Actress runtime The runtime support is realized as a thin wrapper over the Akka actor system[22]. It configures the Akka with some default values related to message queues priority settings and fault tolerance. It also registers a shutdown hook to gracefully terminate the running adaptive elements, giving them the opportunity to execute the `destroy` method.

5.3 Verification Support

Flawed self-adaptive software may cause serious damages[23] and therefore they should undertake a rigorous validation and verification process. One of the advantages of using models for defining feedback control loop architectures is that it makes them amenable for automated analysis and verification. Figure 5.4 shows a high level overview of the verification support in Actress. It consists of two components that will be discussed in this section: model consistency checking and external model verification.

5.3.1 Model Consistency Checking

Model consistency checking is concerned with determining whether models are consistent according to their specifications [Kolovos et al., 2006]. These specifications are usually expressed as constraints in the form of structural invariants, *i.e.*, boolean expressions that

[22] http://doc.akka.io/docs/akka/2.2.0/general/actor-systems.html

[23] For example, in September 2010, Facebook faced an outage for more than two hours caused by feedback control system. According to facebook spokesman they had entered a feedback loop that did not allow the databases to recover. http://edition.cnn.com/2010/TECH/social.media/09/24/facebook.outage/index.html

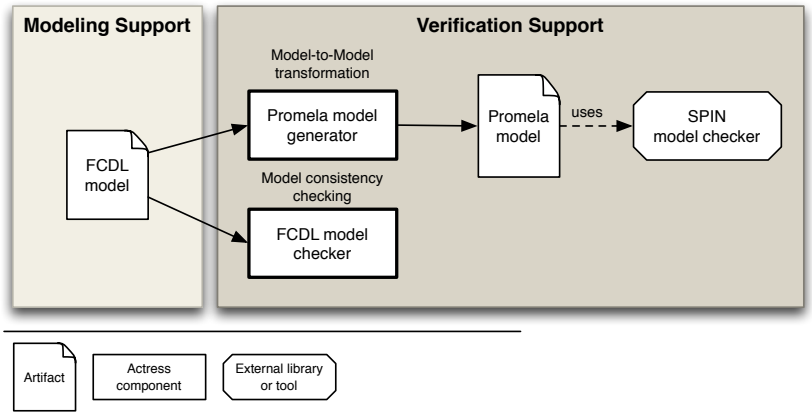


Figure 5.8: Overview of Actress validation and verification support

query model instances checking whether particular properties are satisfied. Model invariants are commonly supported in EMF-based models by either Java through Eclipse Validation Framework[24] or by more specialized languages such as *Object Constraint Language* (OCL) [Object Management Group, 2012b] or *Epsilon Validation Language* (EOL) [Kolovos et al., 2009]. Additionally, as a part of this thesis we have developed an alternative approach based on a dedicated internal DSL in Scala that is further described in the second part of the thesis (*cf.* Chapter 7).

Invariants are used in the FCDL meta-model for asserting the model well-formedness. For example, definitions presented in Chapter 4 are implemented as model constraints (*cf.* Section 7.2). Additionally, developers can define their own set of invariants for FCDL model instances. Usually, they are used to identify architecture bad smells such as adaptive element overlaps (*e.g.* an effector being orchestrated by multiple controllers). Using OCL, these invariants can be directly attached to the model as annotations and checked automatically by the EMF validator. For example, we could attach an OCL[25] annotation to `LoadMonitor` element to ensure that the `size` and `requests` ports are connected to two different sources (to prevent mistakenly connecting them to the same `Accumulator` for instance):

```
@OCL(invDifferentSource="
  self.ports
    ->select(p | p.name = 'size' || p.name = 'requests') // select ports
    ->collect(p | p.connections) // select their connects
    ->collect(p | p.parent) // select owning instances
    ->asSet()->size() == 2 // there must be two different ones
  ")
processor LoadMonitor {
```

[24] http://www.eclipse.org/modeling/emf/?project=validation

[25] The type of the `self` is `AdaptiveElementInstance` since the OCL annotation is attached to `AdaptiveElementType` that is instantiated to `AdaptiveElementInstance` and the attached expressions are evaluated for adaptive element instances.

5.3.2 External Verification

The use of models and MDE techniques brings the possibility of external model verification. It is based on model transformations, whereby an input model (*i.e.* FCDL) is transformed into some others, usually formal models. The transformed models are then checked by some external model checking tools in order to verify that some desired properties hold. Concretely, we use this technique to verify assumptions about connectivity and reachability properties using Promela and the SPIN model checker [Holzmann, 2003].

For example, we can check at design time which elements will be activated by data emission of a given element. Since all the tools that are built on the top of this model must support the same semantics, we can ensure that these properties hold across all the generated artifacts. In the running scenario, for example, one might want to ensure that the `utilController` will be activated when there is a new access log record. Using the LTL formulae [Huth and Ryan, 2004] this invariant can be express as:

$$\Box\,(\,\texttt{accessLogParser}_{activate} \rightarrow (\Diamond\,\texttt{utilController}_{activate}))$$

The predicate variables $\texttt{accessLogParser}_{activate}$ and $\texttt{utilController}_{activate}$ relate to the `accessLogParser` and `utilController` elements and are **true** when the respective activate methods (`activate`) are executed. This expression means that: ``*always* ($\Box$) when the `accessLogParser` `activate` interaction contract is executed, then the `utilController` `activate` interaction contract will *eventually* ($\Diamond$) be executed as well.'' These properties can be easily verified by the SPIN model checker. SPIN takes a model of the system described in a Promela modeling language. It will try to find a counter example in which the negated LTL formulae holds proving the corresponding stack trace. The Promela model can be generated from our architecture model simply by mapping the element interaction contracts and message flow into the corresponding Promela concepts. For example, the `accessLogParser` which is an instance of `AccessLogParser` whose interaction contract is $\langle \Uparrow (\texttt{lines}); \emptyset; \Uparrow (\texttt{requests}, \texttt{size}) \rangle$ is translated into Promela code shown in Listing 5.2.

```
1  // act activate(lines; ; size, requests)
2  #define accessLogParser_activate (accessLogParser@act_activate)
3
4  // ports
5  chan accessLogParser_port_lines = [1] of { mtype };
6  chan accessLogParser_port_size = [1] of { mtype };
7  chan accessLogParser_port_requests = [1] of { mtype };
8
9  active proctype accessLogParser() {
10   // which interaction has been executed
11   byte act = 0;
12
13   // infinite process
14   end:
15
16   // waiting for activation
17   waiting:
18     if
19       // act activate(lines; ; size, requests)
20     :: accessLogParser_port_lines ? PUSH -> act = 1;
21     fi;
```

```
22
23   // element activations
24   executing:
25     if
26       // act activate(lines; ; size, requests)
27     :: act == 1 ->
28       act_activate:
29         accessLogParser_port_size ! PUSH;
30         accessLogParser_port_requests ! PUSH;
31     fi;
32
33   act = 0;
34   goto waiting;
35 }
```

Listing 5.2: Example of the generated Promela code for `accessLogParser`

The transformation between FCDL and Promela is driven by adaptive elements interaction contracts (*cf.* Section 4.2) and by the model of computation (*cf.* Section 4.1). The process starts with a main composite. Each element instance (the transformation is working on an instance level) is mapped into an active Promela process and each port instance into a buffered channel. Additionally, there are channels for all provided sensors that allow their containing elements to simulate propagation of state changes. The communications over these channels follow the adaptive elements interaction contracts.

Based on the notion of interaction contracts, a similar verification support is also presented in the work of Cassou *et al.* [Cassou et al., 2011].

5.4 Integrated Development with Actress

So far we have presented the Actress components individually. In this section we will put them together and discuss their integration inside the Eclipse IDE and how they cover the whole process of self-adaptive software system integration.

The modeling environment is built on the top of the Eclipse platform and it integrates with the Eclipse IDE. It leverages from Eclipse eco-system to covers the main tasks related to development of FCDL-based self-adaptation inside a unified development environment (*cf.* Figure 5.9):

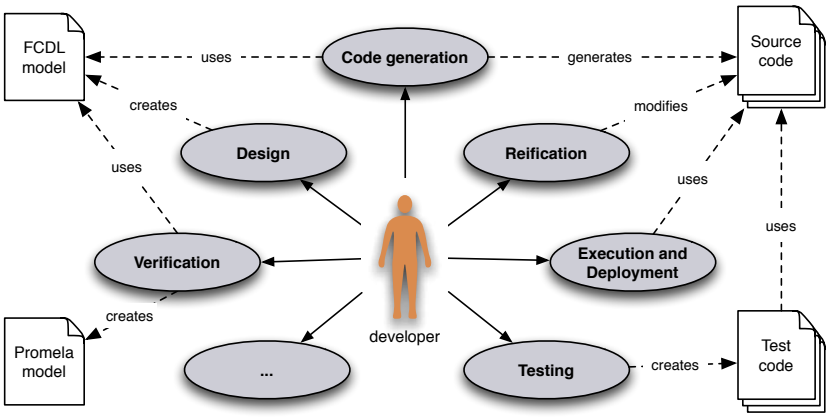


Figure 5.9: The main tasks and artifacts involved in the development process with Actress

– *Design.* For designing feedback control architectures using FCDL we provide the xFCDL language with the Xtext based programming support (*cf.* Section 5.1). Figure 5.10 shows a screenshot of the xFCDL editor in action.

– *Verification.* A developer can provide his own set of FCDL structural invariants in a variety of languages including OCL, Java or Scala. These languages provide a solid Eclipse integration and can be directly edited and executed from within Eclipse. Similarly, generated Promela models can also be directly reviewed and executed inside Eclipse through either EpiSpin[26] or EP4S[27] plug-ins.

– *Code Generation.* Every time a user saves an edited xFCDL file, the Actress code generator (*cf.* Section 5.2) generates all corresponding sources. They can be inspected using in the regular Eclipse Java editor.

– *Reification.* The developer has to provide custom implementation of adaptive elements that do not use Xbase expressions. The advantage of Eclipse is that one can easily mix multitude of languages inside one project inside the same development environment. Regardless of what programming language the developer chooses, the IDE allows to mix them providing a comparable editing experience and interconnection between

[26] http://epispin.ewi.tudelft.nl/
[27] http://matrix.uni-mb.si/en/science/tools/eclipse-plug-in-for-spin/

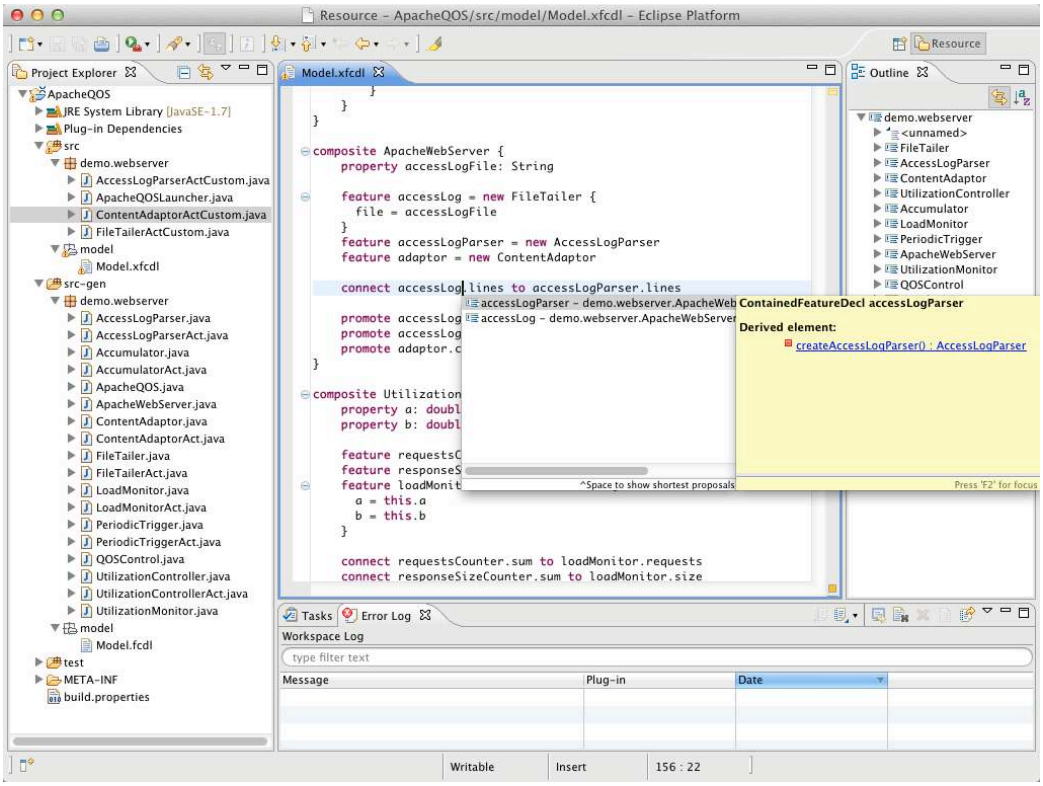


Figure 5.10: The Actress modeling environment

them (*e.g.* one can easily navigate and debug across classes written in different languages). This support is possible since all languages in the end share the same JVM model.

– *Execution and Deployment.* The code generator creates Java launchers for the FCDL main composites and thus they can be launched directly from within Eclipse. Thanks to JSR 45[28] (debugging support for other languages), Xtext leverages from the Eclipse Java Debugger and allows to directly debug the Xbase expression that are embedded inside domain-specific languages. For deployment, currently the support relies on the Eclipse ability to export executable JAR files from a project.

– *Testing.* Since the generated code is a plain Java code it can be simply tested using standard JUnit tests leveraging from Eclipse JUnit support for test launching.

5.5 Summary

This chapter concludes the first part of this thesis that is about self-adaptive software systems integration. Concretely, the chapter has introduced the Actress modeling environment and provided a detailed discussion about its support for modeling, code generation and verification. For the modeling support, it presented xFCDL, a domain-specific

[28] http://jcp.org/en/jsr/detail?id=45

language for defining FCDL architectures including adaptive elements implementation using the Xbase expression language. We then discussed a code generation and the actor-based execution environment followed by the support for FCDL verification. Finally, putting the individual Actress parts together we have discussed the integrated development capabilities and the integration of Actress modeling environment into the Eclipse IDE.

Essentially, Actress consists in a series of model manipulations taking models as inputs and producing other models or text artifacts, or asserting models consistency. In the next part we are going to discuss model manipulations and detail the particular techniques we used for the Actress implementation.

Part II

On Model Manipulation

CHAPTER 6

Model Manipulation Tasks, Languages and Tools

In the second part of this thesis we focus on model manipulation languages as the underlying facility for implementing the Actress modeling environment. A number of different external Domain-Specific Languages for model manipulation have been proposed. They allow developers to manipulate models using higher-level abstractions resulting in gains in expressiveness and ease of use when compared with the use of general purpose programming languages. On the other hand, the complexity of their provided tool support and performance limitations together with language usability and interoperability issues can significantly contribute to accidental complexities as we have experienced during the Actress development. In this and the next chapter, we explore an alternative approach to model manipulation based on internal DSLs. Using Scala as the host language, we implemented a model consistency checking and model transformations internal DSLs and use them for Actress implementation.

We start with an overview of model manipulation languages, present the major approaches to EMF model manipulation and discuss some of their shortcomings. Finally, we outline the internal DSL approach and elaborate on the host language selection.

6.1 Motivation

This section introduces the context of model manipulation clarifying our reasons for seeking alternatives to current approaches. The second part provides an overview of the model consistency checking and model transformations tasks.

6.1.1 Why an Alternative to Model Manipulations?

Model manipulation languages and tools play an essential role in Model-Driven Engineering by providing support to automate operations such as model consistency checking,

transformation, comparison and merging [Sendall and Kozaczynski, 2003]. These tasks are used to gradually refine abstract model concepts into concrete implementations. For example, we have shown that Actress uses model-to-model transformations to translate xFCDL model into FCDL (*cf.* Section 5.1.5) and JVM (*cf.* Section 5.1.4) models that are then used to synthesize actor-based applications (*cf.* Section 5.2.1) using model-to-text transformation. At each phase of this transformation chain, source models consistency have to be checked so the transformation executes and the resulted artifacts are not flawed.

In the past years, different technologies have been proposed to facilitate these tasks, particularly within the EMF. Since the EMF code generator maps the model concepts to Java (*e.g.* a class in a model becomes a Java class, an attribute becomes an instance variable), developers can use it directly to manipulate any sort of EMF models. However, a loss of abstraction occurs as a result of mapping the modeling concepts into Java, and the lack of certain language constructs such as higher-order functions and type inference makes the code far from being clean and concise. This eventually gives rise to accidental complexities [Schmidt, 2006]. The inability of some GPLs to alleviate this complexity by expressing domain concepts effectively [Schmidt, 2006] is one of the reasons why a growing number of different model manipulation DSLs have been proposed. These DSLs allow developers to manipulate models using higher-level abstractions resulting in gains in expressiveness and ease of use [Mernik et al., 2005].

There has been a considerable amount of effort invested in a number of EMF model manipulation languages. For example the OMG standard implementation family with OCL [Object Management Group, 2012b] for model consistency checking, QVT [Object Management Group, 2011a] and MOFM2T [Object Management Group, 2008] for model transformations; the Epsilon project [Paige et al., 2009] that builds an extensive family of model manipulation DSLs based on a common expression language; Kermeta [Muller et al., 2005] with a single, but more general imperative language for all model manipulation tasks; and ATL [Jouault and Kurtev, 2006], a popular model-to-model transformation language. These *external* DSLs are built from the ground up and thus they provide specific constructs that allow developers to manipulate models using higher-level abstractions. While the external DSL approach allows these languages to be independent and offers complete design freedom, evolving complex external DSLs to a sufficient degree of maturity requires significant effort [Steele, 1999, Chafi et al., 2010]. It requires not only domain knowledge and language development expertise, but also considerable engineering effort, particularly to provide solid tool support such as IDEs, debuggers and profiles [Mernik et al., 2005, Chafi et al., 2010]. Furthermore, maintaining the infrastructure for these DSLs is also mobilizing a lot of resources. It is hard to get to an industrial strength level, but even harder to keep it. Tool support and performance are then easily jeopardized. Moreover, a DSL usually does not exist in isolation and thus features such as access to native objects, code reuse across language families and composition of manipulation tasks, must be provided and maintained.

Some of these concerns have been raised by others [Kolovos et al., 2006, 2008b, Chafi et al., 2010] and we have directly observed them while developing Actress. They give

rise to accidental complexities, albeit of a different nature to those associated with the use of GPLs. This together with the fact that most of the Actress code is performing model manipulation has led us to look for alternative approaches.

We identified the shortcomings as being related to scalability issues in terms of flexibility and performance of both DSLs and their tool support. In particular, model manipulation languages contain a lot of general programming facilities which make the languages rather complex. Therefore, we explore an alternative approach in which we embed model manipulation constructs to a GPL.

An internal DSL leverages the constructs and tools of its host language and thus less effort is needed to develop and evolve the DSL and its tool support. For this to be an effective approach the abstractions provided by the underlying GPL must be flexible enough to support definition of domain-specific constructs that can be conveniently supported by the GPL tool set. We use Scala, a statically typed object-oriented and functional programming language that notably provides a flexible syntax, higher-order functions, implicit definitions and mixin-class compositions. Together with the seamless integration with EMF, this allows one to realize type-safe and consistent internal DSLs with significantly reduced engineering effort. Moreover, we can leverage the highly usable tool support provided by major tool vendors as well as from the many existing Java and Scala libraries.

6.1.2 Model Manipulation Tasks

Model manipulation covers a wide range of languages and tools automating different tasks such as model consistency checking, model transformation, model comparison and model merging. In this work, we focus primarily on model consistency checking and model transformations as they are the key tasks enabling model-driven software development [Sendall and Kozaczynski, 2003] and the most important for the Actress implementation.

Model Consistency Checking Determining whether a model is consistent according to its specifications is one of the most sought model management operation [Kolovos et al., 2006]. Model consistency checking provides facilities to capture structural constraints as state invariants and to check model instances against these constraints. It complements the limited expressiveness of the structural constraints of the meta-modeling languages where it is usually narrowed to containment and type conformance [Kolovos et al., 2009]. Essentially, a structural constraint is a boolean query that determines whether a model element or a relation between model elements satisfies certain restrictions.

Model-to-Model (M2M) Transformation Another frequently performed task in MDE is M2M transformation. Sendall and Kozaczynski [Sendall and Kozaczynski, 2003] even refer to it as ``*the heart and soul of model-driven software development*''. Essentially, it takes as input one or more source models conforming to some meta-models and convert them to one or more target models conforming to (not necessarily different) meta-models [Sendall

and Kozaczynski, 2003]. Among the number of different styles, *declarative*, *imperative*, *operational* and *hybrid* are the most implemented ones [Kolovos et al., 2008b]. Declarative style is based on a rule-based pattern matching of source model elements and is generally suitable for transformation scenarios where the source and target meta-models have similar structure. Imperative style on the other hand addresses well cases, in which sequential processing and complex mappings are required. However, it is usually described at a low-level abstraction leaving developers to deal manually with issues like resolving transformed elements from their source counterparts and the overall transformation orchestration. Operation style is similar to imperative but offers some more dedicated M2M transformation constructs [Czarnecki and Helsen, 2006]. Finally, hybrid languages integrate declarative and imperative transformation styles providing a declarative rule-based execution support with imperative constructs.

Model-to-Text (M2T) Transformation Similarly to M2M transformation, M2T transformation takes one or more source input models, but instead of other models, it outputs textual artifacts [Rose et al., 2012] (*e.g.* application implementation and documentation). The main approach followed by M2T transformations is to use templates whereby the target text contains placeholders for data to be extracted from the source model [Czarnecki and Helsen, 2006, Object Management Group, 2008]. The escape direction in templates defines which part of the template shall be escaped. Usually, templates employs *text-explicit* style in which text producing logic is escaped using a pair of delimiters. However, in cases where the amount of text-producing logic outweighs the text being produced (*e.g.* source code generation), it is more convenient to write the logic directly and escape the static text parts instead using *code-explicit* templates.

6.2 Approaches to Model Manipulation

Among the different languages and tools that can be used for EMF model manipulation we have selected the following technologies that support model consistency checking and model transformation tasks: Xtend[1], Eclipse implementations of OMG standards, Epsilon, ATL, and Kermeta. These languages were selected because:

- they include a variety of approaches a GPL (Xtext), families of DSLs (OMG, Epsilon), a dedicated task-specific language (ATL) and a more general yet still a domain-specific language (Kermeta);
- they also represent different development approaches including industrial implementation of a proprietary language (Xtext), language standards (Eclipse / OMG), but also academic research (Epsilon, ATL, Kermeta);
- they are actively developed and used projects with an established user community; and
- they have influenced our approach.

[1] http://xtend-lang.org

Figure 6.2 shows a matrix of the selected languages and the model manipulation tasks they concern.

		Model Consistency Checking	Transformation M2M	Transformation M2T
Eclipse OMG	OCL	✓		
	QVTo		✓^{O}	
	Acceleo			✓^{T}
Epsilon	EVL	✓		
	ETL		✓^{H}	
	EGL			✓^{T}
ATL			✓^{H}	
Kermeta		✓	✓^{I}	✓^{C}
Xtend		✓	✓^{I}	✓^{C}

H: Hybrid, O: Operational, I: Imperative M2M transformations.
T: Text-explicit, C: code-explicit M2M transformations.

Figure 6.1: Summary of surveyed model manipulation languages

OMG Model Manipulation Standards The Eclipse Modeling project hosts implementation of some of the OMG model manipulation standards that are usable with EMF models. Eclipse OCL[2] provides an implementation of the Object Constraint Language for model consistency checking, Eclipse QVTo[3] implements the operational component of the QVT standard for M2M transformation, and Acceleo[4] is an implementation of the MOFM2T M2T transformation standard. These projects have been developed from the ground-up, including custom infrastructure for the IDE support. The Eclipse OCL has been later re-engineered and is now based on Xtext[5]. Both QVTo and Acceleo are reusing OCL for model navigation. QVTo further introduces *ImperativeOCL* as an imperative language extending conventional OCL by expressions with side-effects [Büttner and Kuhlmann, 2008]. The MOFM2T standard defines both text- and code-explicit styles of M2T transformation, but Acceleo only implements the former one.

Epsilon Epsilon [Paige et al., 2009] is both a platform providing an extensive family of model management languages and tools, and a framework for creating new DSLs by exploiting the existing ones. It is currently one of the most complete language workbench

[2] http://www.eclipse.org/modeling/mdt/?project=ocl
[3] http://projects.eclipse.org/projects/modeling.mmt.qvt-oml
[4] http://www.eclipse.org/acceleo/
[5] http://xtext.org

for model manipulation tasks. Among others, it includes the following task-specific languages: *Epsilon Validation Language* (EVL) for model consistency checking [Kolovos et al., 2009], *Epsilon Transformation Language* (ETL) for model-to-model transformation [Kolovos et al., 2008b], and *Epsilon Generation Language* (EGL) for model-to-text transformation [Rose et al., 2008]. All these languages are based on a common OCL-like expression language called *Epsilon Object Language* (EOL) [Kolovos et al., 2006]. EOL is an imperative dynamically typed language supporting many of the OCL features.

ATL One part of the *Atlas Model Management Architecture* (AMMA) [Kurtev et al., 2006] platform is the *Atlas Transformation Language* (ATL), a hybrid M2M transformation language. It is one of the most used language for M2M transformation to date [Kolovos et al., 2008b]. It can also do other model manipulation tasks such as model validation [Bézivin and Jouault, 2005], although the language is not particularly suitable for it [Kolovos et al., 2008b]. The ATL language is conceptually built on a subset of OCL for both its data types and its declarative expressions.

Kermeta A different approach to model manipulation is undertaken by Kermeta [Muller et al., 2005]. It is an object-oriented imperative language that can be used to implement any model manipulation task without emphasizing on any particular one. It allows specifying static semantics and model checking using OCL, meta-model prototyping, simulation and implementation for model operations to be weaved directly into meta-models. Kermeta is strongly typed and compiles into Scala code.

Xtend Graduating from *OpenArchitectureWare* (OAW) framework[6], Xtend has evolved into general purpose programming language that itself presents as a Java dialect. It comes with a comprehensive tool support that is built on the top of the Xbase/Xtext framework. It includes features such as extension methods, lambda expressions and operator overloading. It is a type-safe language that compiles directly into Java code. Similarly to Kermeta, it can be used to implement any model manipulation task and it provides a notable support for string template expression[7] making it very convenient for M2T transformation[8]. Furthermore, while it does not provide any particular support for model consistency checking, it can use Xtext validators[9], a Java-based API for declarative model constraints.

6.3 Issues in Model Manipulation Languages

In the previous section we have shown a number of different languages for model consistency checking and model transformations. Since typically MDE involves a series of independent model manipulation tasks, the multitude of languages raises issues concerning

[6] http://www.openarchitectureware.org/

[7] http://www.eclipse.org/xtend/documentation.html#templates

[8] Xtend 2 replaces Xpand which was the M2T transformation language in OAW (http://blog.efftinge.de/2010/12/xtend-2-successor-to-xpand.html)

[9] http://www.eclipse.org/Xtext/documentation.html#validation

language interoperability, consistency and reuse [Kolovos, 2008]. These accompany the general quality attributes related to expressiveness and versatility as well as to tool support and performance. The following subsections discuss theses issues in the light of the presented model manipulation languages.

It is important to note here that the following study is not supposed to be an extensive survey of issues of model manipulation languages. Rather, it presents a summary of the major shortcomings encountered with these technologies. Many of the problems have been acknowledged by the community and we also report on works in which similar problems were discussed. However, despite the fact that many of these issues have already been identified and some of them also addressed, the lack of an overall integration yet remains a major issue.

6.3.1 Language Expressiveness and Versatility

Expressiveness Following are some of the main issues regarding the expressiveness of these languages:

– *Model Navigation.* Essentially, any model manipulation technique is based on a set of basic facilities for model *navigation* and *modification* [Kolovos et al., 2006]. Besides Xtend, all the other model manipulation languages are built, at least conceptually, on some subset of OCL for model navigation. One of the key points that Anders Ivner mentions with regards to OCL in the foreword to the second edition of *The Object Constraint Language* [Warmer and Kleppe, 2003] is ``*Second, it is compact, yet powerful. You can write short and to-the-point expressions that do a lot*''. While this is true for many of the short and straight-forward expressions, when the complexity grows the ease of reading and writing of these expressions decreases radically. OCL queries are defined in form of expressions that can be chained together. The underlying linearity of this chaining often leads to long and complex expressions that are difficult to understand and maintain. To address this Correa *et al.* [Correa et al., 2009] provide some refactoring techniques to simplify some of these complex expressions. Ackermann *et al.* [Ackermann and Turowski, 2006] propose to utilize specification patterns for which OCL constraints can be generated automatically, together with a collection of OCL specification patterns formally defined. These techniques might help in certain cases, but in general there is no simple solution to break the linearity of the expressions. Since both Acceleo and ATL are based on OCL they are also affected by this issue (QVTo is using ImperativeOCL that supports statement sequencing). EOL alleviates this issues, by adding support for statement sequencing. On the other hand, it is a dynamic language and as such there is only limited support for static checking resulting in typing errors occurring at runtime (*e.g.* accessing a non-existent field or operation). Both Kermeta and Xtend support statement sequencing.

– *Invariants Capturing Constructs.* There are some problems with invariants capturing constructs offered by OCL. Kolovos *et al.* [Kolovos et al., 2009] give a detailed overview of these issues, proposing EVL language to address them. They identify the following

missing features: *support for user feedback* to additionally provide a meaningful message to the user, *support for different levels of severity* indicating the level of a constraint violation, *support for constraint dependency* preventing meaningless evaluation of constraint whose dependency are not satisfied, *flexible context definition* grouping together related constraints under the same dependency, and *support for inconsistency repair* offering means to repair the inconsistency by suggesting changes in the affected model elements similarly to the ``*quick fixes*'' known from the modern IDEs. From our experience in Actress development we add *support for invariant reuse* across different contexts and different models, so they do not have to be copied and pasted.

Furthermore, OCL can only use one meta-model at a time, which is particularly limiting for inter-model consistency checking.

Since Kermeta support of model consistency checking mimics the OCL one, it is also affected by the same issues (besides the single model access). Moreover, in Kermeta, there is no guarantee for side-effect free expressions.

Xtend implements structural invariants as regular methods and thus they can contain side-effects code. It also does not support dependent constraints, context definitions nor inconsistency repairs.

– *Separation of Concerns* Regarding M2T transformation, a potential issue with MOFM2T represented by Acceleo is that it couples transformation content with its physical output [Rose et al., 2008]. The output destination is a first-level language concepts and thus additional work is required to pipe the transformation to other destinations than physical files (*e.g.* HTTP responses, sockets).

Versatility In general, Domain-specific languages provide more-specific, but less versatile language constructs that in some case might present certain limitations.

For example, OMG with ATL and Epsilon inherits insufficient OCL support for template types. In EMF, this results in the inability to work with generic types[10] besides generic collections.

Apart from Xtend, all other languages are limited in the number of operations that support in their standard libraries. For example, OCL does not support regular expressions. Since this would be rather limiting particularly for model transformations, QVTo, Acceleo and ATL extend OCL string by adding regular expression support. All the languages also support accessing native code, *i.e.*, Java methods, however in the cases of Kermeta or OCL the interoperability is not straightforward[11].

Xtend supports most of the Java concepts, but it is missing support for constructing anonymous[12] and inner[13] classes which might be problematic for interfacing with some existing Java libraries.

[10] The new pivot-based Eclipse OCL supports the template types, but it is due to be integrated across the other languages, *e.g.* https://bugs.eclipse.org/bugs/show_bug.cgi?id=397429

[11] For example in OCL: http://bit.ly/1bKxCu8, in Kermeta: http://bit.ly/1bKxvi0

[12] https://bugs.eclipse.org/bugs/show_bug.cgi?id=402388

[13] https://bugs.eclipse.org/bugs/show_bug.cgi?id=376399

6.3.2 Interoperability, Consistency and Reuse

Interoperability We have already mentioned that model manipulation tasks are seldom carried out alone, rather they are chained together in a form of model manipulation workflows. While all the mentioned languages are able to access the same type of models, *i.e.*, EMF models, they usually do not provide any support for modeling workflows and thus each task has to undergo model loading and storing. In the case of large models these can become time and resource consuming operations. There is no interoperability in this sense between the OMG languages. Epsilon solves this problem by proposing a set of Ant[14] tasks for orchestrating different Epsilon task-specific languages without unnecessary model loading or storing [Kolovos et al., 2008a]. However, in Epsilon there is no direct support[15] for embedding model manipulation tasks within one another. For example to invoke a M2T transformation from within a M2M transformation (*e.g.*, to synthesize method body while transforming an adaptive element type into a JVM class as it is done in Actress). This is supported in both Kermeta and Xtend since all model manipulation tasks are expressed within the same language so they also share one runtime model. On the other hand neither of them provide any particular support for loading and storing EMF models and they only rely on the EMF API.

Consistency Regarding OCL, there is a number of inconsistencies and confusions that have already been reported [Kolovos et al., 2006, Wilke and Demuth, 2010, Willink, 2011a, Cabot and Gogolla, 2012] and that we encountered during our development: different symbols to navigate through sets (`->`) and scalars (`.`); implicit `oclAsSet()` when applied onto **null** object; implicit collection flattening when using `collect()` operation; and logical operators with `undefined` values.

The same issues also appears in QVTo, Acceleo and ATL. Moreover, while these languages are based on OCL and thus their expressions looks similar, there is nothing like a standard OCL core that these languages could built on [Kolovos et al., 2006] and thus there are some misalignments in the subset of OCL implemented. Some operations are only available in one of the languages or under different names. For example, each of the languages uses a different name for accessing model element container: `oclContainer()` (OCL), `container()` (QVTo) and `eContainer()` (Acceleo). Another example is the aforementioned regular expression support that is additionally implemented by QVTo, Acceleo and ATL. Since it is not driven by any standard, the operations and their names differs (*e.g.* `replaceFirst()` (QVTo), `replace()` (Acceleo), no corresponding operation in ATL).

Reuse Kolovos *et al.* [Kolovos et al., 2008b][16] identified code reuse issue among OMG languages and ATL. The main problem is that while all these languages allow one to define auxiliary operations, they cannot be reused among them. For example, in the Actress

[14] `http://ant.apache.org/`

[15] Epsilon can access native object and thus its own Java API, however there is no direct support in the language.

[16] This issue has been identified in 2008, but it has been only now (2013) raised with Eclipse OCL and OMG `http://www.eclipse.org/forums/index.php/t/458134/`

transformation chain, each of the model manipulation needs to check whether an FCDL element contains specific annotations. Using OCL, Acceleo and QVTo/ATL we would have to define the same help function three times (*cf.* Listing 6.1). Despite that these languages are similar and the actual expression is the same, code duplication introduces a maintenance problems and is a source of potential code errors.

```
-- OCL
context ModelElement
 def: hasAnnotation(name : String) : Boolean =
   self.annotations->exists(st|st.name = name)

-- QVTo
query ModelElement::hasAnnotation(name: String) : Boolean {
 return self.annotations->exists(st|st.name = name)
}

-- Acceleo
[query hasAnnotation(elem : Element, name : String) : Boolean =
 elem.annotations->exists(st|st.name = name) /]

-- ATL
helper context ModelElement def: hasAnnotation(name: String):
 Boolean = self.annotations->exists(st|st.name = name);
```

Listing 6.1: Helper function in OCL, QVTo, Acceleo and ATL

Apart from the Epsilon family that is based on the same common language and runtime, all the other languages define their own syntaxes and runtimes and thus the code cannot be easily reused among them. The only reuse is possible through their Java APIs, which as stated above, is not always the most convenient to use.

6.3.3 Tool Support and Performance

There is a close relationship between a language and its supporting tools. Often a language choice is influenced by the accessibility of appropriate tools that facilitate the language use. However, providing a usable and mature tool support requires significant engineering effort. Next to compiler infrastructure and libraries, it also includes all the other aspects of modern development tooling such as rich editing support, debuggers, profilers, unit testing support, integration into build tools and execution performance.

All of the surveyed languages provide Eclipse-based tooling with different features and degrees of maturity. Figure 6.3.3 gives an overview of the tool support and language infrastructure of the selected technologies.

Testing in Epsilon is supported by a dedicated language [García-Domínguez and Kolovos, 2011] and IDE support, Kermeta includes JUnit like framework and Xtend being based on Java supports all Java unit test frameworks. There is a mixture of interpreted and compiled languages. The runtime performance is further discussed in Section 8.3.2 on page 190.

Regarding the performance of the supporting development tools, in some cases we found some impediments, notably with OCL, Acceleo and Kermeta. The editors of the former two start having difficulties when editing files with more than 1000 lines[17]. In

[17] Tested with Eclipse Juno on MacBook Pro 2.53 Ghz Intel i5, 8GB RAM

	OCL 3.2.2	QVTo 3.3.2	Acceleo 3.3.2	Epsilon 1.0.0	ATL 3.3.1	Kermeta 2.1	Xtend 2.3.1
Code assists	++	++	++	+	++	++	++
Refactoring	-	-	+	-	-	-	+
Static checking	++	++	++	+	++	++	++
Hyperlinking	++	++	++	+	-	-	-
Debugging	-	++	++	++	++	++	++
Profiling	-	++	++	-	++	-	+[1]
Testing	-	-	-	++	-	+	+
Build tool integration	-	-	M,A	A	A	M	M,A
Execution	I/Cj[2]	I	I	I	Cc	Cs	Cj

++ : strong support, + : basic / restricted support, - : no support
M: Maven, A: Ant build tool integration
I: Interpreted, Cc: Compiled into custom byte code
Cj: Compiled into Java, Cs: Compiled into Scala
[1]supported using Java profilers, [2] since Eclipse Juno [Willink, 2012]

Figure 6.2: Model manipulation tool support comparison

case of OCL this issue is related to its transition to Xtext which reduced the grammar by factor of 5, but the generated parser is 10 times larger and 11 times slower [Willink, 2011b]. Kermeta compiles to Scala after every keystroke taking a significant amount of resources. All the languages allow one to access native Java objects, but besides Xtend, there is no possibility to debug the code while debugging the language itself (these debuggers runs on the interpreted code). Finally, while Epsilon alleviates many of the other approaches shortcomings, the choice of dynamic language makes it difficult to provide more advanced tooling support.

6.4 Towards Internal DSL

This last section presents our motivation behind internal DSLs, highlights the main design principles and derives the requirements for the host language. These rules are then used to choose a suitable host language with an overview of its main features for DSL embedding.

6.4.1 Rationale

In the case of model manipulation DSLs, implementation effort of both the language and tool support is very laborious since they provide a lot of general purpose functionality (*e.g.*, loops, functions, object creations, inheritance) making the languages rather complex. Therefore, instead of embedding general-purpose constructs into a model manipulation DSL, we explore an alternative internal DSL approach whereby the specific model manipulation constructs are embedded into a GPL.

An internal domain-specific language[18] is represented within the syntax of some GPL with a stylized usage of that language for domain-specific purposes [Fowler, 2010]. One of the advantages of an internal DSL is the support of separation of concerns. The programming languages and compiler experts build GPLs while domain experts build DSL and not the general purpose features [Siek, 2010]. Furthermore, the engineering effort is significantly reduced as the host language infrastructure is reused, including not only the compiler and the runtime, but also features supporting IDE integration, debugging and profiling support.

On the other hand, being embedded in a host language imposes certain limitations to the DSL design. The most apparent problem besides syntax limitations is that the DSL becomes a leaky abstraction [Siek, 2010], *i.e.*, it does not reduce, nor hide the complexity of the host language. For example, the abstraction layer formed by an internal DSL becomes completely transparent during debugging. The underlying host language is not hidden from the developer and the scope of the DSL can be easily escaped making it traditionally difficult to perform domain-specific analyses and optimization [Chafi et al., 2010]. There are trade-offs between the full flexibility in external DSLs on the one hand and reduced engineering effort of internal DSLs on the other hand. Yet, with a host language that delivers both flexibility and extensibility together with highly-usable tool support, we could provide similar expressiveness of the existing model manipulation languages, but with less effort.

6.4.2 Principles

Internal DSLs are built using general programming language constructs and they operate at the same level of abstraction as the host language. Therefore, designing an internal DSL consists of mapping the domain-specific concepts into the concepts of the host language such as user-defined functions, classes and data structures. What makes an internal DSL difficult is to find the right balance between clearly expressing a domain concept yet in a form that fits the target language *look-and-feel* [Wyk et al., 2007]. The focus should be to reuse the existing language infrastructure and libraries and only add the missing pieces though extensions.

– *Designed as a Library.* The internal DSLs should not require any major change in the host language environment configuration. Each model manipulation language should be designed as a library and using it should be just a matter of importing that particular library. This will also make it possible to compose different DSLs together as well as to build custom higher abstractions on top of them in a structured way.

– *Having a common infrastructure.* All the external model manipulation DSLs presented in Section 6.2 are based on some common infrastructure that provides basic model manipulation operations such as model navigation and modification. Similarly, the internal DSLs should share a common base facilitating the model navigation and modification.

[18] also known as *Domain-Specific Embedded Language* (DSEL)

It will also provide a general support for automating tasks that do not fall into the patterns targeted by task-specific languages.

– *Similar to existing DSLs.* As we have already discussed in Section 6.3, all the presented languages are based at least conceptually on some subset of OCL for the model navigation and constraints expressions. Besides it shortcomings [Krikava and Collet, 2012a], OCL has a significant user base and its navigational expression are considered expressive. Therefore the navigation facilities should resemble OCL expressions and include model querying operations such as the OCL `select()` or `collect()`.

  For the particular model manipulation tasks the internal DSLs should support the established concepts and use the same terminology. The aim is to improve the productivity by allowing users to write model manipulation tasks in a similar fashion as in the existing model manipulation DSLs, but without being constrained with the less versatile language constructs and limited IDE support.

– *Competitive Performance*. The performance differences, both in speed and in memory footprint, between a model manipulation task written in an internal DSL should be comparable to the external DSLs.

6.4.3 Choosing the Host Language

Some languages are more suitable for DSL embedding than others. Based on our motivations and the determined principles, following are the requirements that we find the most important for the host language:

– *Highly usable tool support.* The tool support should be highly usable and well integrated with the target meta-modeling environment, *i.e.*, with Eclipse IDE. The usability of a language tool support is difficult to define since it largely depends on the preferences and background of its users. However, having a rich editing support (*e.g.*, content assists, refactoring, hyperlinking, outline view, static checking) as well as the support for dynamic program analysis through debugging and profiling is known to improve productivity [Pfeiffer and Wasowski, 2012].

– *DSL support.* The chosen host language should support DSL embedding. It should have *flexible* syntax allowing to express the domain-specific constructs that are present in the existing model manipulation languages. It should be also *extensible* to allow to add missing constructs and operations in the way that they have the same *look-and-feel* of host language constructs.

– *Static type safety with type inference.* While type safety might be a matter of preference, we believe that due to the type checking it increases model manipulation productivity. Using type inference provides type safety without the necessary syntax clutter. Moreover, static typing also enables precise content assist in the text editor.

– *Functional tendencies.* The expressive power of OCL comes from the declarative nature of the language, support for higher-order functions in collection operations, tuples, and

explicit collection initialization [Akehurst et al., 2008]. The language should therefore support the same so that OCL-like expressions could be easily implemented.

– *Runtime compatibility*. The host language must be compatible with the target metamodeling runtime platform. In the case of EMF, the host language has to run on the JVM platform.

– *Competitive performance*. The host language should be efficient in terms of performance simply because we cannot provide any sort of efficient internal DSLs if the host language is of a poor quality. Considering JVM platform, the performance between the host language and Java should be negligible.

6.4.4 Why Scala?

From the many programming languages available on the JVM platform[19], we identify the following ones that satisfy the above requirements to some degree: Scala, Xtend, Groovy[20], Cylon[21]. Cylon supports all our requirements, but it is a rather a new language that has yet not been finalized. Groovy is a mature language, but despite the fact that it partially supports type checking, it still remains primarily a dynamic language[22]. We have already mentioned Xtend (*cf.* Section 6.2 on page 130) as a GPL that can be used for model manipulation. Next to Scala, it supports all the requirements, but similarly to Cylon, the language is still under development and its support to internal DSL is mostly limited to extension methods and operator overloading. We therefore find Scala to be the most suitable general-purpose programming language for our model manipulation internal DSLs.

Scala is a statically typed mature GPL with type inference that combines both object-oriented and functional style of programming with advanced tool support. It is interoperable with Java, has an extensive collection library, and it has been designed to host internal DSLs [Chafi et al., 2010]. It provides support for lifting static source information, allowing one to customize error messages in terms of the domain-specific extensions [Moors et al., 2012] and provides more advanced constructs for deep DSL embedding [Rompf and Odersky, 2010].

6.4.5 Embedding DSL in Scala

A typical way of embedding a shallow DSL into Scala is by designing a library that allows for writing fragments of code with domain-specific syntax. The fragments will be weaved within Scala own syntax so that it appears as being different [Dubochet, 2011]. There are four essential Scala features that we rely on for the DSL embedding:

– Flexible syntax that permits omitting semicolons and dots in method invocations, as well as infix operator syntax for method calls.

[19] Wikipedia lists more than 50 languages http://en.wikipedia.org/wiki/List_of_JVM_languages
[20] http://groovy.codehaus.org/
[21] http://cylon-lang.org
[22] http://groovy.codehaus.org/Type+checking+extensions

– Support for implicit type conversions allowing one to extend existing types with new methods. For example, we can extend the existing `Boolean` type with an `implies` operation that is not part of the Scala standard library:

```
// 'a' represents the instance being extended by this class
implicit class ExtendedBoolean(a: Boolean) {
 // 'b' is call-by-name parameter
 def implies(b: => Boolean) = !a || b
}
```

With this extension imported and thanks to the flexible syntax we can use `implies` in a natural way, keeping the language *look-and-feel*:

```
val a = true
val b = false
val c = a implies b // false
```

Implicit conversions also provide a convenient way for extending existing modeling classes with auxiliary informations that cannot or should not be part of the meta-model.

– Support for *call-by-name* parameters, *i.e.*, parameters that are not evaluated at the point of function application, but instead at each use within the function (*e.g.* the `b` parameter from the `implies` operation is only evaluated when `a` is **true**).

– Support for *mixin-class composition* allowing to reuse the partial class definition in a new class [Odersky et al., 2004, Section 6.1]. For example, the above definition of `ExtendedBoolean` could be placed into a trait:

```
trait MyExtensions {
 // ...
 implicit class ExtendedBoolean(a: Boolean) { /* ... */ }
 // ...
}
```

which when mixed-in gives an easy access to all the definitions:

```
class MyClass extends MyExtensions {
 // ...
 val c = a implies b
}
```

6.5 Summary

This chapter presented the context of model manipulations in EMF. It has identified 3 main model manipulations tasks and presented 9 popular languages that represent a spectrum of model manipulations approaches. Some of their shortcomings were discussed and finally the motivation and principles behind our internal DSL approach including the requirements and selection of the host language were presented. The very last subsections have shown the basic Scala features for DSL embedding.

The next chapter illustrates how these Scala features and some others can be used to develop internal DSLs for EMF model manipulations.

CHAPTER 7

Lightweight Model Manipulation Using Sigma

Following the previously developed principles, this chapter presents a family of model manipulation DSLs called Sigma. It is a Scala library that implements internal DSLs for model consistency checking, model-to-model and model-to-text transformations. For each language the requirements as well as the syntax and semantics are presented. The language features are presented gradually using real examples taken from the Actress code base. Some technical details about how certain DSL constructs are implemented are deliberately skipped and instead a reference to further documentation is given.

We start with developing a common infrastructure for model navigation, modification and delegation on which all the DSLs are built. Next, we give an overview of the task-specific languages starting with the model consistency checking DSL, followed by the M2M and M2T transformation DSLs.

7.1 Model Navigation, Modification and Delegation

Any model manipulation technique is in one way or another based on a set of basic facilities for model navigation and modification. This section presents basic model operations that provide a common infrastructure layer on which the task-specific model manipulation languages are built. It also includes model delegation allowing one to implement EMF derived properties and operation bodies directly in Scala.

7.1.1 Requirements

The common infrastructure connects a particular meta-modeling technology to a host language, that is, mapping the modeling concepts into objects and classes. In the case of EMF, it involves aligning EMF generated Java classes with Scala, facilitating navigation (*i.e.* extracting information of interest) and modification operations (*i.e.* creating, updating and

deleting models and model elements). The main requirement for both navigation and modification facilities is to provide similar expressiveness to the one found in OCL and in the other model manipulation languages.

7.1.2 Model Navigation

Following is the overview of the model navigation support in Sigma.

Basics In OCL, retrieving names of all adaptive elements annotated with the `Main` annotation can be expressed using the following query[1]:

```
system.types
  ->select(e | e.annotations->exists(a | a.name = 'Main'))
  ->collect(e | e.name)
```

Expressing the same in Scala using only the generated Java classes is cumbersome. First, EMF uses Java beans like getters to access model elements structural features that results in a ``*get noise*'' issue (*e.g.* `system.getTypes`). Second, EMF collections (`EList` and `EMap`) extends Java collections that only support iteration through imperative control structures.

To address these limitations, in addition to the EMF code generator, for each EMF model element we generate a set of extensions that makes the Java classes more interoperable with Scala. For example, following is the generated extension trait[2] for the FCDL types package (*cf.* Appendix B.2):

```
trait TypesPackageScalaSupport extends EMFScalaSupport {
  implicit class ControlSystemScalaSupport(that: ControlSystem) {
   // for each property generate Scala getter
   def types = that.getTypes
   def dataTypes = that.getDataTypes
   // ...
 }
}
```

Mixing this trait allows one to navigate the model without the get prefixes (*e.g.* `system.types`). The trait extends an `EMFScalaSupport` trait that implicitly converts EMF collections into corresponding Scala collections (*i.e.* `EList` becomes Scala `Buffer` and `EMap` becomes mutable Scala `Map`). The conversion is based on the Scala facilities to convert between Scala and Java collections and it only happens at the interface level leaving the underlying data storage unchanged. Furthermore, it extends some existing Scala types with missing OCL operations such as the boolean `implies` operation (*cf.* Section 6.4.5). As a result, the expression is almost identical to the OCL one:

```
system.types
  .filter(e => e.annotations exists (a => a.name == "Main"))
  .map(e => e.name)
```

[1] The variable `system` is an instance of `ControlSystem` (*cf.* Appendix B.2), which is the container of all adaptive element types.

[2] Scala traits are similar to classes but they can include methods implementations and be used in mix-in composition [Odersky et al., 2004]

Pattern Matching Scala supports pattern matching that can be used in combination with partial functions. For example, listing all main composites links can be concisely expressed as[3]:

```
system.types collect {
  case c: CompositeType if c.annotations->exists(_.name == "Main") => c.links
}
```

The `collect` operation performs collection filtering (the `case` and `if` part) and mapping (`=>`) at the same time. It accepts a partial function that uses pattern matching to select the interesting instances and map them to new results. This makes the resulting query easier to write and read than its OCL counterpart:

```
system.types
  ->select(e | e.oclIsKindOf(CompositeType)) -- only Composites
  ->collect(e | e.oclAsType(CompositeType)) -- type to CompositeTypes
  ->select(e | e.annotations->exists(a | a.name = 'Main')) -- only 'Main'
  ->collect(e | e.links) -- select links
```

The pattern matching can be exploited even further. For example, following is the code that selects all adaptive elements push input ports and pull output ports:

```
system.types map (_.ports) collect {
  case p @ Port(_, INPUT, PUSH) => p
  case p @ Port(_, OUTPUT, PULL) => p
}
```

First, we select all ports `system.types map (_.ports)` and then using pasttern matching we further filter the respective ports types and modes. The use of `Port(_, INPUT, PUSH)` relies on extractors that decompose object properties into tuples and the `p@Port(...)` is a variable binding. An extractor is just a method (`unapply`) that can be included in the generated trait:

```
trait TypesPackageScalaSupport extends EMFScalaSupport {
  implicit class PortScalaSupport(that: Port) {
    object Port {
      // extractor method decomposing object properties
      def unapply(that: Port): Some[(String, PortType, PortMode)] =
        Some((that.getName, that.getPortType, that.getPortMode))

      // ...
```

The pattern patching is particularly useful for M2M transformations (*cf.* Section 7.3.2).

Handling Nulls In general, a query can result in invalid or undefined values. Such case appears when an unset reference is navigated. For example, converting adaptive element name into uppercase (*e.g.* `name.toUpperCase`) might result in a runtime exception (in all presented languages including Scala) in case the name property has not been set yet. While it is trivial to fix, the extra check for non-nullity is easy to forget since there is nothing in the language itself that forces one to do so. However, the information that a property

[3] In this example we use an underscore which is a placeholder for parameters in anonymous functions instead of specifying parameters names.

is optional is already contained in the meta-model, expressed as a $0..1$ multiplicity. The problem is that the EMF generated method signature corresponding to this attribute accessor does not take the optionality into account (it is only mentioned in the generated documentation). It simply generates the same method as in the case of a $1..1$ multiplicity:

```
public interface NamedElement extends ModelElement {
  // ...
  String getName();
  // ...
}
```

The generated Scala getters, on the other hand, can take this into account and wrap optional features into Scala `Option`, a container explicitly representing an optional value:

```
implicit class NamedElementScalaSupport(that: NamedElement) {
  def name: Option[String] = Option(that.getName)
  // ...
}
```

As a consequence it is no longer possible to access the feature value directly and the new type forces one to always check for the presence of the value. The idiomatic way is to treat the `Option` type as a monad[4], which in the case of the name example results in following expression[5]:

```
self.name map (_.toUpperCase) getOrElse "NO NAME"
```

It is important to note here, that even a required attribute might still be **null**. However, this fact will be discovered during model consistency checking that usually precedes all other model manipulations.

7.1.3 Model Modification

The aim of model manipulation is to provide facilities allowing a seamless creation, update and removal of model elements. By design, OCL does not have model modification capabilities, but in EOL for example, a new adaptive element type can be created using:

```
var elem = new AdaptiveElementType;
elem.name = "MyComposite";
elem.active = false;

var aPort = new Port;
aPort.name = "aPort";
aPort.portType = PortType#INPUT;
aPort.portMode = PortMode#PUSH;
elem.ports.add(aPort);
```

In Sigma, instances of model elements can be created using the following code:

```
val elem = AdaptiveElementType(name = "MyElement")
elem.active = false
elem.ports += Port(portType = INPUT, portMode = PUSH, name = "aPort")
```

[4] http://www.scala-lang.org/api/current/index.html#scala.Option

[5] The _ is used as a placeholder for parameters in the anonymous functions

An expression such as `AdaptiveElementType(...)` is translated into an invocation of an `apply` method on the `AdaptiveElementType` object, *i.e.*, `AdaptiveElementType.apply(...)`, which is basically a counterpart of the extractor. We generate this method into the package support trait together with a compatible setters for each changeable feature.

These methods provide a convenient way to author complete models directly in Scala. Additionally, we provide a support for delayed initialization, for the cases when the initialization of an element should only happen after its containment, and lazy resolution of contained references[6]. Both facilities are realized using the standard EMF notification mechanism and EMF adapters [Steinberg et al., 2008, Section 16.2].

7.1.4 Model Delegation

EMF model elements can define operations, derived properties and derived references. The EMF code generator creates their corresponding method signatures whose implementation is left to the developer to complete. These methods are placed in the very same classes as all the other generated code. There are several well-known problems with this approach[7]:

- complicated merge between generated and non-generated code,
- non-generated code is *lost* among the generated code,
- generated code has to be put under version control,
- unneeded files are not removed (*e.g.* classes of removed elements from model are not deleted),
- easy to forget to remove the `@generated` annotation that results in code losses,
- implementation code must be done in Java.

To address these issues, Sigma embraces the generation gap pattern [Fowler, 2010] and separates EMF generated files from the handwritten ones. Part of Sigma is an executable class that invokes the amended EMF code generator[8]. The modified code generator generates all EMF classes into a separate source folder, *i.e.*, `src-gen`. Furthermore, it changes the default EMF factory to instantiate EMF delegate classes instead of the default implementation. For example, consider xFCDL `AdaptiveElementDecl`. Its EMF classes will be generated into

- `src-gen/.../AdaptiveElementDecl.java` and
- `src-gen/.../impl/AdaptiveElementDeclImpl.java`.

If there exists a file `src/.../AdaptiveElementDeclImplDelegate.scala`, the xFCDL factory will be changed to use this class instead of the default one:

```
public AdaptiveElementDecl createAdaptiveElementDecl()
{
  AdaptiveElementDeclImplDelegate adaptiveElementDecl =
    new AdaptiveElementDeclImplDelegate();
```

[6] Technical details with examples are available at http://fikovnik.github.io/Sigma/model-navigation-and-modification.html

[7] http://www.slideshare.net/meysholdt/codegeneration-goodies

[8] This approach has been inspired by Xtext MWE http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)

```
  return adaptiveElementDecl;
}
```

The `AdaptiveElementDeclImplDelegate` extends the original `AdaptiveElementDeclImpl` and so developers can easily provide implementation of derived properties and operations bodies using Scala and Sigma model navigation and modification support developed in the above sections.

```
1 class AdaptiveElementDeclImplDelegate extends AdaptiveElementDeclImpl {
2   // implementation of derrived reference
3   override def properties = that.declarations collect {
4     case x: PropertyDecl => x
5   }
6   // ...
7 }
```

7.2 Model Consistency Checking

This section provides an overview of the Sigma internal DSL related to model consistency checking.

7.2.1 Requirements

Similarly to OCL and EVL a structural invariant should be expressed as a boolean query and the language should provide first class support for all the additional EVL features (*cf.* Section 6.3.1): user feedback, different levels of severity, constraint dependency, flexible context definition, inconsistency repair as well as for invariant reuse.

7.2.2 Model Consistency Checking in Scala

The internal DSL for model consistency checking is based on the concept of a validation context. A validation context has a name, defines the type of the model element it checks, and groups together a set of invariants. We illustrate it on the example of checking interaction contracts consistency (*cf.* Definition 5). The example is built gradually, presenting the features one by one.

Validation Contexts A new validation is created by extending `ValidationContext` trait:

```
1 class InteractionContractValidationContext extends ValidationContext
2   with InvMethods with TypesPackageScalaSupport {
3
4   type Self = InteractionContract // define context type
5 }
```

In this example, the class definition also mixes in the `InvMethods` trait that provide method-based style representing constraints as Scala methods. Other styles will be discussed later in the next subsection. It also mixes in `TypesPackageScalaSupport` trait, which is the generated common infrastructure for the FCDL types model (*cf.* Appendix B.2). Line 4

specifies the context type of the validation context, *i.e.*, the type of instances the invariants are going to be applied to. An invariant is simply a Scala method:

```
def invDataRequirementsHaveOnlyPullInputPorts = self.requirements forall { p =>
  p.portType == INPUT && p.portMode == PULL
}
```

This method defines an invariant *DataRequirementsHaveOnlyPullInputPorts* verifying that the data requirement part of an interaction contract contains only pull input ports. By convention, an invariant is a method that starts with `inv` prefix, takes no parameters and returns either a boolean or an instance of `ValidationResult` (*cf.* next paragraph). `InvMethods` is using reflection to find all such methods at runtime. Similarly to OCL and EVL, `self` represents the current instance that is being checked. Internally, it is a getter defined in the validation context base class.

User Feedback and Severity Levels Instead of returning a boolean, an invariant method can return `ValidationResult` that provides an opportunity to specify a detail message and a severity level. It is an abstract class that can be concretely instantiated as either `Passed` (validation passed), `Canceled` (validation has been canceled, this case is discussed later in the subsection), `Warning` (validation did not pass, but it is not a critical situation), or `Error` (invariant failed). For example:

```
def invDataRequirementsHaveOnlyPullInputPorts = {
  val res = self.requirements forall { p =>
    p.portType == INPUT && p.portMode == PULL
  }

  if (res) Passed
  else Error("Data requirements can only define pull inputs ports")
}
```

This way, in the case the invariant fails on some interaction contracts, users will be provided with a more friendly error message than the generic one (*The 'DataRequirementsHaveOnlyPullInputPorts' constraint is violated on 'InteractionContract(...)'*).

Quick Fixes The customized error message already gives a better feedback, however, it still does not provide enough information, *i.e.*, it does not say which port(s) are affected. The way the invariant is defined is essentially a direct translation from the definition: $R = \Downarrow (in_1, \ldots, in_n)$, where $in_i \in I^{\Downarrow}$. An alternative way, which leads to a better user feedback, is to do the opposite and look for an counter example, *i.e.*, a port that is not an input port or whose mode is not pull: $\exists in \in R$ such that $p \notin I^{\Downarrow}$ (*cf.* Definition 5). If there is such a port we can improve the feedback by not only a more precise message, but also by providing a quick fix that might remove the problem. Concretely, we can refine the invariant to the following:

```
1 def invDataRequirementsHaveOnlyPullInputPorts = {
2   self.requirements find { p => p.portType != INPUT || p.portMode != PULL } match {
3     case None => Passed
4     case Some(p) =>
5       Error(s"Port '${p.name}' is not an pull input port.")
```

```
6       .quickFix(s"Remove '${p.name}' from interaction contract data requirements") {
7         self.requirements -= p
8       }
9       // ...
10    }
11  }
```

The expression on line 2 checks for the counter example. If none is found (line 3) the invariant has passed. If there is one (line 4), an error message is created (line 5) containing the name of the concerned port together with a quick fix (lines 6-8).

Figure 7.1 displays a mock of the screen showing Sigma model consistency checking in the Actress modeling environment. For this to work, the error has to define additional

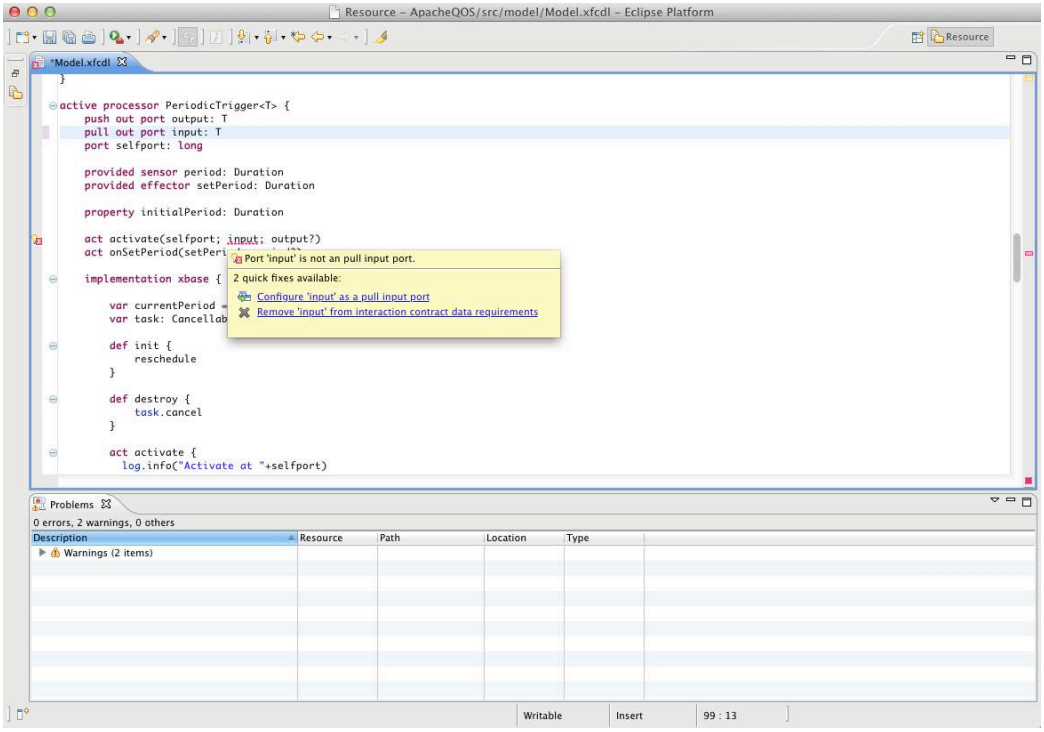


Figure 7.1: A mock of Sigma model consistency checking in Actress

meta-data information such as the index of the problematic element and the meta-object of the corresponding EMF reference [Steinberg et al., 2008].

Constraint Guards and Dependencies The data requirements definition of interaction contracts also states that each pull input port shall be presented at most once. This can easily be turned into a new invariant[9]:

```
def invDataRequirementsHaveUniquePorts = {
  self.requirements.distinct.size == self.requirements.size
}
```

[9]For brevity we use the shortest implementation.

However, it might only make sense to check this once the other data requirement invariant has been satisfied. This is supported by adding an invariant guard, *i.e.*, a boolean expression that is evaluated before the invariant itself:

```
def invDataRequirementsHaveUniquePorts = guardedBy {
  self satisfies invDataRequirementsHaveOnlyPullInputPorts
} check {
  self.requirements.distinct.size == self.requirements.size
}
```

An invariant that fails because of unsatisfiable guard condition returns `Canceled` result.

Context Guards Similarly to an invariant, a validation context can also have a guard that further limits its applicability. For example, we may decide to only check interaction contracts that are not annotated with `@Ignore`. Such a validation context guard is expressed as:

```
1  class InteractionContractValidationContext extends ValidationContext
2    with InvMethods with TypesPackageScalaSupport {
3
4    type Self = InteractionContract // define context type
5
6    // validation context guard
7    override def guard = !(self.annotations exists (_.name == "Ignored"))
8
9    // ...
10 }
```

Constraint Reuse Many of the FCDL model elements define a name attribute. Most of them do that by extending from `NamedElement`. Similarly, there are many xFCDL elements also defining a name attribute. However, they do not extend from the same superclass. Yet, it would be useful, if for both models we could share the same name attribute invariants. One way to do this is to extract the invariant code and define it externally and then to simply delegate to it from corresponding validation contexts. While, this is a reasonable option in some cases, it still introduces code duplication. Since `ValidationContext` is a Scala trait, instead of a delegation, multiple validation contexts can be mixed in. For example:

```
1  trait NameValidationContext extends ValidationContext with InvMethods {
2    // context is defined using Scala structural type
3    type Self <: { def getName(): String }
4
5    // shared invariant
6    def invNonEmptyName = self.getName.nonEmpty
7  }
8
9  class InteractionContractValidationContext extends ValidationContext
10   with InvMethods with TypesPackageScalaSupport with NameValidationContext {
11     // ...
12 }
```

On line 1 a validation context trait is defined for any modeling element that has a name property. It uses Scala structural typing [Odersky, 2011, Structural Types on page 171]

(line 3). A structural type defines only class structure but does not associate it with any concrete type. This trait is then mixed in with our original `InteractionContractValidationContext` on line 10.

Internal Representation Internally, `ValidationContext` represents all the constraints as a set of nested `Constraint` classes. Together, these two classes form the abstract syntax of the consistency checking DSL. The mixed-in `InvMethods` trait provides one concrete syntax, representing constraints as methods. Despite the strict boundaries of the internal DSL syntax imposed by the host language, multiple different syntax styles can be constructed. For example, a trait `InvBuilder` provides a builder style syntax for creating structural constraints:

```
class InteractionContractValidationContext extends ValidationContext with InvBuilder
 with TypesPackageScalaSupport {

 constraint("DataRequirementsHaveOnlyPullInputPorts")
  .check {
   val res = self.requirements forall { p =>
    p.portType == INPUT && p.portMode == PULL
   }

   if (res) Passed
   else Error("Data requirements can only define pull inputs ports")
  }

 constraint("DataRequirementsHaveUniquePorts")
  .guardedBy { self satisfies "DataRequirementsHaveOnlyPullInputPorts" }
  .check {
   self.requirements.distinct.size == self.requirements.size
  }
}
```

Both classes result in the same abstract representation.

Execution Since validation context is just a Scala class, it can be easily combined in any model manipulation workflow. To use it is just a matter of instantiating and invoking its `validate` method. Another option is to register the Sigma EMF validator that is configured with all relevant validation contexts and then use the standard EMF validation mechanisms [Steinberg et al., 2008, Chapter 18].

7.3 Model Transformations

This sections presents the internal DSLs for M2M and M2T transformations.

7.3.1 Requirements

Requirements elicitation for model transformation languages is a challenging task since the variability of this domain is rather large [Czarnecki and Helsen, 2006]. Similar to Kermeta and Xtend, an imperative style of M2M transformation is already supported due to our common infrastructure layer described in Section 7.1. On the other hand, as we

have discussed earlier, the lower-level of abstraction of the imperative transformation style leaves users to manually address issues such as orchestrating the transformation execution and resolving target elements with their source counterparts. Therefore, inspired by both ATL and ETL, we provide a dedicated internal DSL that combines the imperative features with declarative rule-based execution scheme into a hybrid M2M transformation language.

In M2T transformation, we focus on a template-based approach. There is a range of possible textual outputs that can be generated from a model. The main difference is the ratio between the amount of text fragments and code driving the transformation. For example generating source code will most likely contain more code than generating HTML documents. In model-driven software development, the aim is to synthesize running systems implementation and therefore our primary focus is source code generation.

Similarly to the model consistency checking section, the internal DSLs are gradually presented on an example taken from Actress code base.

7.3.2 Model-to-Model Transformation

The M2M transformation DSL organizes transformations in classes. It supports transforming an arbitrary number of input models into an arbitrary number of output models. We illustrate the DSL on xFCDL to FCDL translation (*cf.* Section 5.1.5).

Syntax Transformation rules constitute the abstract syntax of the M2M transformation DSL. Similarly to ETL or ATL, a rule defines a source and a target element to which it transforms the source. It can optionally define additional targets, but there is always one primary source to target relation. A rule can also be declared as `lazy` or `abstract` using the respective `@Lazy` or `@Abstract` annotations. Similarly to consistency checking constraints, transformation rules can optionally limit their applicability by defining a guard, making it a conditional rule.

The following code excerpt shows the basic structure of a Sigma M2M transformation.

```
class XFCDL2FCDL extends M2M with RuleMethods
  with XfcdlPackageScalaSupport with TypesPackageScalaSupport {

  // register target models
  val targetMetaModels = Seq(TypesPackage.eINSTANCE)

  // transformation rule
  def ruleAdaptiveElement(source: AdaptiveElementDecl, target: AdaptiveElementType) =
    guardedBy {
      // guard block
      source.role != COMPOSITE
    } transform {
      // transformation body
      target.name = source.name
      target.active = source.active

      // ...
    }
}
```

The transformation is a Scala class that extends `M2M` base class (line 1) together with the generated packages support traits (for both xFCDL and FCDL meta-models) for model navigation and modification (line 2). A convenient way to represent transformation rules is by using methods. The `RuleMethods` trait provides the necessary support for that, using reflection to find all the rules at runtime. For example, the `ruleAdaptiveElement` method denotes a transformation rule that for a given xFCDL adaptive element declaration produces an FCDL adaptive element type (line 8). It has a guard (line 9) limiting its applicability to only non-composite adaptive element declarations and a body (line 11) where elements attributes are copied (lines 14-15).

Rule Signatures The `ruleAdaptiveElement` method above shows one out of three possible method signatures for M2M transformation rules:

– *Regular rule* lists the target model elements as method parameters:

`def` rule$Name$`(source: S, target`$_1$`: T`$_1$`, ..., target`$_n$`: T`$_n$`): Unit`

`def` rule$Name$`(source: S, target`$_1$`: T`$_1$`, ..., target`$_n$`: T`$_n$`): Option[Unit]`

There has to be at least one target parameter defined. The `S` and `T`$_1$ types defines the primary source-target type relation and will be used for matching the rule. Once the rule is matched, the engine is responsible for creating all target elements and supplying them as methods arguments. For example the `ruleAdaptiveElement` above is a regular rule.

– *Explicit target rule* defines only the source element:

`def` rule$Name$`(source: S): T`

`def` rule$Name$`(source: S): Option[T]`

Similarly to the regular rule, the `S,T` pair defined the primary source-target type relation, but in this case the method itself is in charge of creating the target. Within the method body additional targets can be constructed, but in this case, the developer is responsible for their proper containment. Similarly, the second version denotes a conditional rule. For example we could refine the `ruleAdaptiveElement` as target explicit rule:

```
// transformation rule
def ruleAdaptiveElement(source: AdaptiveElementDecl) = guardedBy {
  source.role != COMPOSITE
} transform {
  val target = AdaptiveElementType(name = source.name, active = source.active)
  // ...
  target
}
```

– *Partial rule* defines only the source element and whose body is expressed as a partial function:

`def` rule$Name$ `= partial[S,T]: PartialFunction[S,T]`

Partial rules provide an alternative representation of explicit target rules. They are the same, but they use partial functions for their implementation turning them into conditional rules by default. For example:

```
def ruleAdaptiveElement = partial[AdaptiveElementDecl, AdaptiveElementType] {
  case source if source.role != COMPOSITE =>
    val target = AdaptiveElementType(name = source.name, active = source.active)
    // ...
    target
}
```

Semantics Each rule is executed for all the source elements it applies to unless it is a lazy or an abstract rule in which case it has to be explicitly activated. A rule is applicable to a model element if the rule source type is assignable from the element type. In the case of a conditional rule, a guard can further narrow the applicability of a rule. When a rule is executed, the transformation engine initially creates all the explicitly defined target elements and passes them to the rule that populates their content.

Source Elements Resolution During the M2M transformation, there is often the need to relate the target elements that have been already (or can be) transformed from source elements. For this purpose, the DSL provides a unary operator ~ (tilde) that can be applied to both an instance of an `EObject` and to a collection of `EObjects`. For example, it is used to convert adaptive element type references:

```
1 def ruleAdaptiveElement(source: AdaptiveElementDecl, target: AdaptiveElementType) {
2   // ...
3   // properties
4   target.properties ++= ~source.properties
5   // ports
6   target.ports ++= ~source.ports
7   // ...
8 }
```

On lines 4 and 6 this operator is used to transform xFCDL properties and ports declarations into FCDL representations. For this to work, there have to exist the corresponding transformation rules (*e.g.* `PropertyDecl` → `Property` and `PortDecl` → `Port`). These rules will be consequently executed for each applicable property and port contained in the given adaptive element declaration.

An interesting property of this operator is that, unlike the equivalents in ETL or ATL, it can be type-safe. Concretely, if there is no rule that translates properties declarations into FCDL properties, the compiler will complain on the line 6 that there is ``*No transformation rule between PropertyDecl and Property*''. This compile time check is realized using implicit values and annotations[10]. It is the same mechanism that is used in Scala collections for polymorphic operations [Moors et al., 2012, Section 4.2]. For this to work, we need to explicitly create the implicit values that encapsulate the transformation rules using a `rule` function:

[10] http://fikovnik.github.io/Sigma/m2m-transformation.html

```
implicit val _ruleProperty = rule(ruleProperty)
```

This function is responsible for converting a method into a transformation rule instance. Furthermore, they also verify that the rule methods are correctly typed, *i.e.*, both parameters and return values are model elements (instances of `EObject`). While a reflection is normally used to introspect the class and automatically register all the rule methods, this way, we gain compile time checking, since the rule applications are resolved already at compile time. The need to explicitly register the transformation rules should be elevated with next version of Scala that will support type macros[11].

Abstract Rules and Rule Extensions Both ATL and ETL support abstract rules and rule extensions. A comparable support is also supported by the internal DSL. For example, next to the adaptive element rule we define a rule for transforming a composite:

```
1  @Abstract
2  def ruleAbstractAdaptiveElement(
3    source: AdaptiveElementDecl, target: AdaptiveElementType) {
4    // transformation of common attributes and references
5  }
6
7  def ruleAdaptiveElement(source: AdaptiveElementDecl, target: AdaptiveElementType) =
8    guardedBy {
9      source.role != COMPOSITE
10   } transform {
11     // delegate
12     ruleAbstractAdaptiveElement(source, target)
13   }
14
15 def ruleComposite(source: AdaptiveElementDecl, target: CompositeType) =
16   guardedBy {
17     source.role == COMPOSITE
18   } transform {
19     // delegate
20     ruleAbstractAdaptiveElement(source, target)
21     // features
22     target.features ++= ~source.features
23     // ...
24   }
```

Using the `@Abstract` annotation, we define an abstract rule that transforms all shared adaptive element attributes and references (line 2). This rule is then explicitly called from both adaptive element rule (line 12) and composite rule (line 20).

Execution Similarly to model consistency checking, M2M transformation is simply executed by instantiating the class and invoking its `apply` method supplying it a source element or an instance of an EMF resource.

[11] In the current 2.0 version type macros are only an experimental feature that is not part of the official language distribution http://docs.scala-lang.org/overviews/macros/typemacros.html

7.3.3 Model-to-Text Transformation

Unlike Acceleo or EGL, the internal DSL for M2T transformation is using the code-explicit form, *i.e.*, it is the output text instead of the transformation code that is escaped. This is one of the syntax limitations that cannot be easily overcome. On the other hand, from our experience, in non-trivial code generations, the quantity of text producing logic usually outweighs the text being produced. For the parts where there is more text than logic we rely on Scala multi-line string literals and string interpolations allowing one to embed variable references and expressions directly into strings.

Syntax and Semantics Similarly to the other M2T transformation languages, the scheduling of template executions is controlled explicitly by the developer starting from the main template. Therefore, there is no execution engine, no additional semantic abstractions and no explicit abstract syntax.

The following code shows an excerpt of the M2T transformation of FCDL model into Promela [Holzmann, 2003] that is used for the external model verification (*cf.* Section 5.3.2):

```
1  class FCDL2PML extends M2T
2    with TypesPackageScalaSupport with InstancesPackageScalaSupport {
3
4    type M2TSource = CompositeInstance
5
6    // the template entry point - main template
7    def execute {
8      // ...
9      genCompositeProcess(source)
10   }
11
12   def genCompositeProcess(elem: CompositeInstance)
13   def genProcess(elem: AdaptiveElementInstance)
14   // ...
15 }
```

Following the same pattern, a M2T transformation is a Scala class extending from the `M2T` base class (line 1). Similarly to the cases shown before, it mixes the common infrastructure for model navigation (line 2). Line 4 defines the model element type that would be the transformation source. A M2T transformation consists in a set of templates that are represented as methods (lines 7, 12, 13). The `execute` method is the entry point, that will be invoked when the transformation is executed. Usually, from there, a transformation is split and logically organized into smaller templates in order to increase modularity and readability.

Text Output Primitives The most common operation in M2T transformation is a text output. The internal DSL provides a several primitives for this. For example, following is the code of the `genProcess` template:

```
1  def genProcess(elem: AdaptiveElementInstance) {
2    !s"active proctype ${elem.name}()" curlyIndent {
3      !s"""
4      // interaction flag (can be 0-${elem.contracts.size})
5      byte act = 0;
```

```
6     // response bit set
7     byte resp = 0;
8     """
9
10    !"end: // infinite process"
11    !"waiting:" indent {
12      !"if"
13      for (ic <- elem.contracts) {
14        indent { !s"// ${ic}" }
15        !"::"
16        indent { genActivationCondition(ic) }
17      }
18      !"fi;"
19    }
20    // ...
21  }
22 }
```

A convenient way to output text in the DSL is through a unary `!` (bang) operator that is provided on strings (*e.g.* line 2). The prefix `s` right before the string double quote denotes an interpolated string, which can include Scala expressions in a type-safe way. Lines 3-8 show example of a Scala multi-line string literal used for longer text blocks that do not involve much text-outputting logic.

An important aspect of any M2T transformation language is the template readability such as layout and indentation. The internal DSL maintains it through dedicated support for decorators, smart whitespace handling and relaxed newlines. Decorators are nestable string operations that reformat a given block. For example, on line 2 we use `curlyIndent` decorator, that wraps its body into a pair of curly brackets and indent each line. Smart whitespace handling removes extra whitespace from multi-line strings that are there only for the template readability. For example the white spaces prefixing the text on lines 3 to 8 will be discarded. Relaxed newlines removes the necessity to output new line characters by doing it automatically after every text output. Both smart whitespace handling and newlines handling are enabled by default, but can be turned off.

Finally, the DSL allows one to fork new text sections. This makes it possible to output text into different locations at the same time. All sections are appropriately merged in the final text at the end of the transformation. This is useful for example for handling imports while generating Java code as they can be resolved one-by-one during the model traversal.

Higher-Level Abstraction Using the text output primitives discussed above, we can build higher-level constructs that encapsulate commonly used patterns. For example, instead of handling indentation every time a common Promela statement (*e.g.*, an if-block or a label) is outputted, we can extract them into separate constructions. The above example (lines 11-19) can be then rewritten as:

```
pml_label("waiting") {
  pml_if {
    for (ic <- elem.contracts) {
      indent { !s"// ${ic}" }
      pml_seq {
        genActivationCondition(ic)
```

```
    }
   }
  }
}
```

The `pml_label`, `pml_if` and `pml_seq` represent constructs for respectively outputting a Promela label, an if-block and a sequence. They are defined as follows[12]:

```
def pml_label(name: String)(block: => Any) {
  !s"$name:" indent {
    block
  }
}
def pml_if(block: => Any) {
  !"if"
  block
  !"fi;"
}
def pml_seq(block: => Any) {
  !"::"
  block
}
```

By defining them in a separate trait, we gain the possibility to reuse them in a different transformations just by mixing-in the trait. Extracting these higher-level constructs only makes sense in certain cases where they can be simple enough not to unnecessarily clutter the API. Since often it is more convenient to output text and rely on the target language infrastructure (*e.g.* compilers) this might provide a reasonable trade-off between a plain M2T transformation and a full M2M transformation [Czarnecki and Helsen, 2006].

Execution Similarly to model consistency checking and M2T transformation, M2M transformation is also executed by instantiating the class and invoking its `apply` method supplying it a source element. In this case the result is a string, that can be stored into an arbitrary destination.

7.4 Summary

This chapter presented a family of model manipulations internal DSLs in Scala called Sigma. It constitutes of DSLs for model consistency checking, M2M and M2T transformations. The DSLs are built on a common generated infrastructure for model navigation, modification and delegation that aligns EMF meta-models with Scala further allowing developers to implement derived properties and operation bodies directly in Scala. Each task-specific language was illustrated on concrete examples taken from the Actress code base.

This chapter also concludes the second part of the thesis concerning model manipulation. The next and the final part focuses on the thesis evaluation and presents conclusions and perspectives for future work.

[12] The `block` method parameters are using the *call-by-name* (`=>`) evaluation.

Part III

Evaluation, Conclusions and Perspectives

CHAPTER 8

Evaluation

Based on the goals identified in Section 1.2 and requirements reified in sections 3.1 and 6.4, this chapter presents the evaluation of the FCDL domain-specific modeling language, the ACTRESS modeling environment and the SIGMA model manipulation DSLs. It is organized into three parts. First we complement the illustration of FCDL on two more case studies based on real-world adaption scenarios. Next, we discuss the first goal of this thesis related to integration of self-adaptive mechanisms into software systems, assessing the suitability of FCDL and ACTRESS. This is followed by the assessment and discussion of the SIGMA internal DSL for model manipulations.

8.1 Experimental Case Studies

Throughout this thesis we have used the QoS management control scenario as a running example to illustrate our approach (*cf.* Section 3.2). In this section we complement the illustration with two additional case studies taken from the domain of *High-Throughput Computing* (HTC). The objective is to provide some additional evidence about the overall feasibility of our approach and a concrete demonstration about the effort required to implement real-world adaptation scenarios with some quantitative evaluation. We aim to demonstrate reuse of adaptive elements across multiple scenarios, hierarchal organization of FCL, and remote distribution of adaptive elements. To achieve this objective, for each case study, we present an end-to-end implementation[1] including the scenario overview, the design of the adaptation engine, feedback control architecture and experimental results accompanied with a discussion.

We start by motivating the chosen case studies and introducing the HTC context. Next, we present a case study addressing overload control in a local batch system which is then extended to cover distributed job submissions.

[1]The supporting material, *e.g.* code and interaction contracts specification is provided in Appendix E.

8.1.1 Why HTC Case Studies?

Context and Motivation High-Throughput Computing (HTC) serves a vital role for researches and engineers by providing them with large amounts of processing power over long periods of time exploiting existing computing resources on the network [Thain et al., 2005]. However, the increasing demand and heterogeneity of tasks to be carried out, and the soaring scale of these distributed environments, make their operation rather difficult and tedious. The practice demonstrates that the human administration cost for HTC infrastructures is high, and end-users are not yet completely shielded from the system heterogeneity and faults [Lingrand et al., 2009, Ferreira da Silva et al., 2012]. Acknowledging the fact that these systems can hardly achieve complete reliability and that they can no longer be controlled statically, new operation modes have to be implemented in order to make them more resilient and dynamically adaptable [Collet et al., 2010].

The following two scenarios address these problems by proposing autonomic operation modes in which the clients adjust their runtime behavior accordingly to the observed state of the environment. Concretely, we consider an HTC environment for executing scientific workflows using the HTCondor[2] infrastructure [Thain et al., 2005]. HTCondor is a well established distributed batch computing system that is being used extensively in both academia and industry. It aims at providing reliable and maintainable high-throughput environments, delivering capacity over long periods of time. The two case studies are motivated by the need to concurrently execute large workflows without overloading the system. In the first scenario we only consider a local job queue and in the second we extend it to use distributed queues.

Including the running scenario (*cf.* Section 3.2), all three case studies are based on a parametric adaptation in the area of client-sever computing. The reason for this is that such cases represent the main adaptation scenarios [Kephart, 2011, Patikirikorala et al., 2012][3].

Alternatives For evaluating the Rainbow framework (*cf.* Section 2.2.2), Cheng *et al.* developed the *Znn.com* benchmark [Cheng et al., 2009b]. While this is a well documented, publicly available benchmark for self-adaptive software systems, we have three reasons for not including it in this thesis: (1) this thesis has been developed in a context of the SALTY project (properly introduced in Section 8.2.1) which focuses on distributed computing infrastructures, (2) the possibilities for adaptation in *Znn.com* are restricted to a parameter-based adaptation of the Apache configuration file and Apache restarting [Tamura et al., 2013]; our running example has already shown a very similar adaptation. (3) *Znn.com* is an imaginary system, while HTCondor is a real system used by many[4]. Nevertheless, providing a FCDL implementation of *Znn.com* is part of our short-term further work (*cf.* Section 9.2.1).

[2] The HTCondor software was known as Condor from 1988 until its name changed in 2012.

[3] More than 50% of 158 papers surveyed by Patikirikorala [Patikirikorala et al., 2012] are in the middleware / data centers domain and more than 40% are concerned with either response time or throughput performance variables

[4] http://research.cs.wisc.edu/htcondor/map/

Next to introducing *Znn.com*, Cheng *et al.* also enumerated a number of requirements that a self-adaptive software benchmark should satisfy. According to the authors: ``*a useful benchmark system is relevant, accessible, dynamically observable and changeable, and versatile; supports alternative adaptive operations and multiple configurations; facilitates quality trade-offs; and can be compared via a common metric*''.

The chosen case studies are addressing a real-world problem of a scheduler overload in an open-source widely used distributed computing infrastructure, HTCondor. The system provides a number of command line utilities to be observed and modified. It contains a wide range of configuration options allowing for various self-adaptive approaches with different trade-offs and configuration paths. Therefore, we believe that the selected case studies provide relevant adaptation scenarios for demonstrating our approach. Moreover, they could serve as the base for developing an alternative to *Znn.com* featuring a real system that has potentially more to offer in the sense of possibilities for self-adaptations.

8.1.2 Case Study 1: HTCondor Local Job Submission Overload Control

Scenario Overview In the HTCondor environment, the default job meta-scheduler that carries out workflow execution is DAGMan. In DAGMan a workflow is defined as a directed acyclic graph where the vertices represent tasks, defined as standard HTCondor jobs, and edges represent the dependencies among them thus specifying the execution order. DAGMan takes a workflow description and continuously submits the ready to be submitted jobs into a local scheduler. This scheduler, called schedd, is responsible for managing a queue of user submitted jobs and for mapping these jobs onto a set of resources where the actual execution is performed. Figure 8.1 shows one of the usual condor set-up [Bradley et al., 2011].

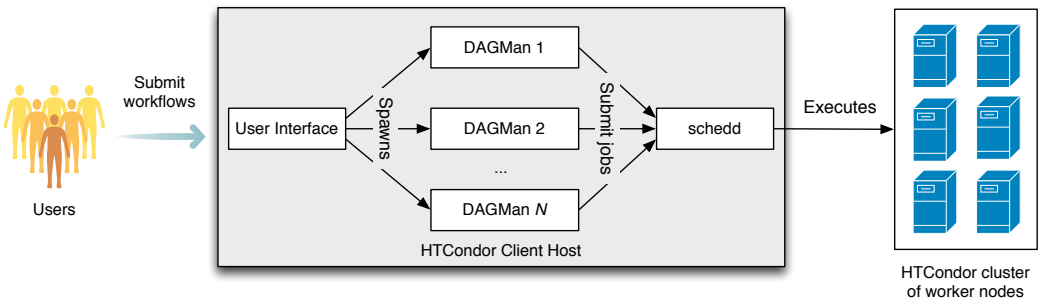


Figure 8.1: Overview of the HTCondor local overload control scenario.

It consists of a client host that provides an interface (*e.g.* a web application) allowing users to submit workflows. Each workflow submission spawns a new DAGMan process which submits workflow jobs into the local schedd. By default schedd accepts all valid submission requests regardless of the current state of the system. Since the work schedd has to perform is proportional to the number of jobs it controls, by executing several large workflows it can become overloaded. This in turn can result in an overall throughput degradation. There is a number of configuration options for fine tuning both the scheduler

and the workflow manager, however, all of them are static and do not consider the current state of the system and its environment.

Adaptation Strategy The task we want to implement is a control system on top of the HTCondor job submission that would ensure high throughput while preventing infrastructure overload. One way to achieve this is to base the adaptation on a model that takes the number of jobs in a queue (N), together with a service rate (μ) and a number of executing DAGMan instances (m) to compute a delay (d) used to throttle DAGMan workflow submission rates and thereby keeping the number of idle jobs within a certain range[5].

In order to maintain a certain utilization of the system and prevent its overload, we design a basic controller that will be integrated in our illustrative architecture. The control maintains a certain number of jobs in the queue, denoted by N^*. There is a configuration option in DAGMan that controls the number of seconds it waits before submitting a task (`DAGMAN_SUBMIT_DELAY`). By making DAGMan to reread this option before each submission, we can impose an adjustable delay, d, for all DAGMans that can be dynamically modified according to the state of the system.

Let m denotes the number of clients representing the running DAGMan instances at some sample time t. Each client i is submitting at rate $\lambda_i = \frac{1}{d}$ therefore the total arrival rate at t is

$$\lambda = \sum_{i=1}^{m} \frac{1}{d} = \frac{m}{d}$$

The control is based on optimizing the utilization of the queue $\rho(N) = \frac{\lambda}{\mu}$ depending on the number of jobs N, in order to maintain N in some interval close to N^*. The model recognizes three states of N:

1. if $N = N^*$ then the queue is ideally filled so we only maintain the $\lambda = \mu$;
2. if it is less, we linearly increase the allowed arrival rate λ; and
3. when it is more we vigorously decrease it all the way to 0 shall $N = N_c$.

This leads to the following control model of the queue growth rate (*cf.* Figure 8.2)[6]:

$$\rho(N) = \begin{cases} \rho_0 + \frac{N(1-\rho_0)}{N^*} & \text{for } 0 < N < N^*, \rho_0 > 1 \\ 1 & \text{for } N = N^* \\ \alpha(N - N_c)^p & \text{for } N > N^*, \text{where } \alpha = \frac{1}{(N^* - N_c)^p} \end{cases} \tag{8.1}$$

where $N_c > N^*$ denotes some critical number of jobs in the queue that must not be reached, ρ_0 is the maximum growth rate allowed in the system and p is a coefficient of the decrease rate in the third case.

[5]There exist two configuration options in HTCondor that express similar concerns. `MAX_JOBS_SUBMITTED` limits the number of jobs permitted in a schedd queue. Submitting a new job over this limit will fail, which may consequently fail the entire workflow execution. `DAGMAN_MAX_JOBS_IDLE` has been recently introduced to DAGMan controlling the maximum number of idle jobs allowed within a workflow before DAGMan temporarily stops submitting. It, however, considers only job clusters (not individual jobs) neither other DAGMans. Both options are expressed as constants in the condor configuration file.

[6]For simplicity, in the first case, we use $N = N^*$ while in practice it would be better to use a vicinity ϵ around N^*

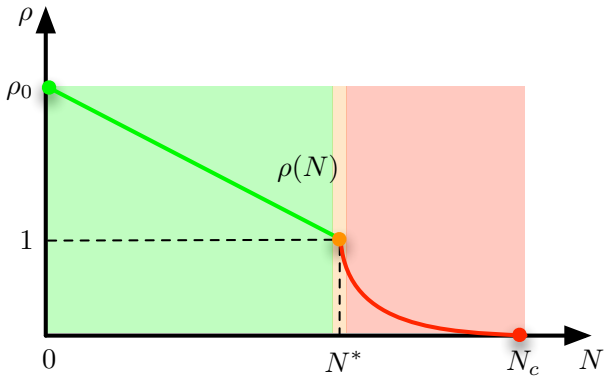


Figure 8.2: Relation between queue utilization ρ and number of idle jobs N

From the utilization and total arrival rate we can derive the target delay d as:

$$\rho(N) = \frac{\lambda}{\mu} = \frac{\frac{m}{d}}{\mu} \qquad d = \frac{m}{\rho(N)\mu} \tag{8.2}$$

Feedback Architecture We start implementing this adaptation scenario by defining the target system touchpoints. First, the following sensors are needed to collect the three metrics (N, μ, m) required by the controller: (1) `CondorQueueStats` that provides the number of idle jobs in the queue N using the `condor_q` command[7], (2) `CondorServiceRate` that computes the current service rate μ from the Condor history file[8] and (3) `ProcessCounter` that obtains the number of DAGMan instances m by executing the system `ps` command[9]. We encapsulate the sensors related to schedd into a composite, `Schedd` (*cf.* Figure 8.3)[10]. They both define `condorConfig` property pointing to the HTCondor configuration file and this way they can share the same property defined at the composite level.

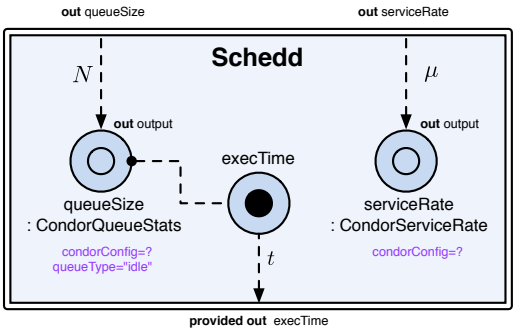


Figure 8.3: `Schedd` composite

The control model (8.1) uses all three measurements at the same time. For this we define a simple `Aggregate` processor. Once pulled over its only output port, it pulls

[7] http://research.cs.wisc.edu/htcondor/manual/current/condor_q.html
[8] http://research.cs.wisc.edu/htcondor/manual/current/condor_history.html
[9] Assuming HTCondor runs on POSIX-like OS, http://pubs.opengroup.org/onlinepubs/9699919799/utilities/ps.html
[10] The `execTime` provided sensor will be explained later in this section.

all connected inputs returning the values synchronized in a tuple. It corresponds to the $\langle \Downarrow (\texttt{output}); \Downarrow (\texttt{queueSize}, \texttt{serviceRate}, \texttt{dagmanCount}); \emptyset \rangle$ interaction contract. Figure 8.4 shows the *Schedd* composite embedded in a new composite, `CondorStats`, together with the `Aggregate` processor and the `ProcessCounter` sensor. To prevent oscillation, we further added `MovingAverage` filters (*cf.* Section 4.1.6). We use them only for the N and μ since the number of HTCondor does not fluctuate much as the workflow are usually long running processes.

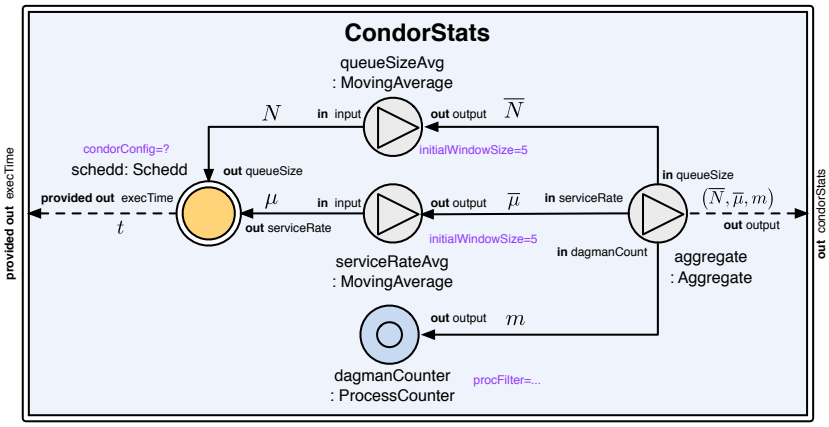


Figure 8.4: `CondorStats` composite.

Next, we need to implement the effector that adjusts the submission delay d. Since DAGMan only evaluates its configuration options when it starts[11], we had to make a small modification that rereads the delay time before each submission from an external file. A `FileWriter` effector is used to simply write the computed delay into that file.

Figure 8.5 shows the resulting architecture, using `LoadController` implementing the actual control model elaborated above (8.1) and (8.2). The `PeriodicTrigger` is the very same as we used for the running example (*cf.* Section 3.3.1).

Hierarchy of Feedback Control Loops - Adaptive Monitoring The primary concern in the above overload control is to protect an unbounded usage of the schedd submission service. The problem is that the protection of one resource might consequently jeopardize another and thus transform one problem into a different and potentially worse one. Therefore, in general, this process has to be done recursively for any probable unbounded resource usage, regardless of whether it is a system resource or a service usage.

In our solution this problem is found in the monitoring part concretely in the `CondorQueueStats` sensor. Internally it executes the `condor_q` command that makes the schedd walks its entire job queue, which becomes an expensive operation when the number of enqueued jobs is large. One way to fix this is by adding a new control layer on top of the existing one that will throttle the execution frequency of the `CondorQueueStats` sensor.

[11] There is a request to change this, *cf.* ticket 2616 (https://htcondor-wiki.cs.wisc.edu/index.cgi/tktview?tn=2616

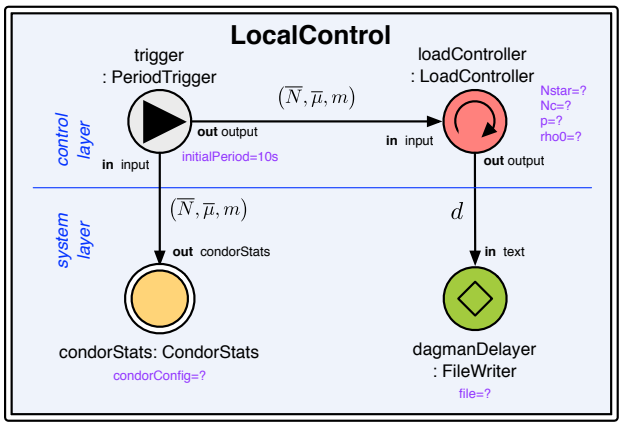
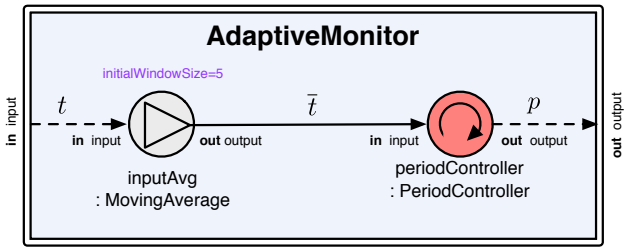
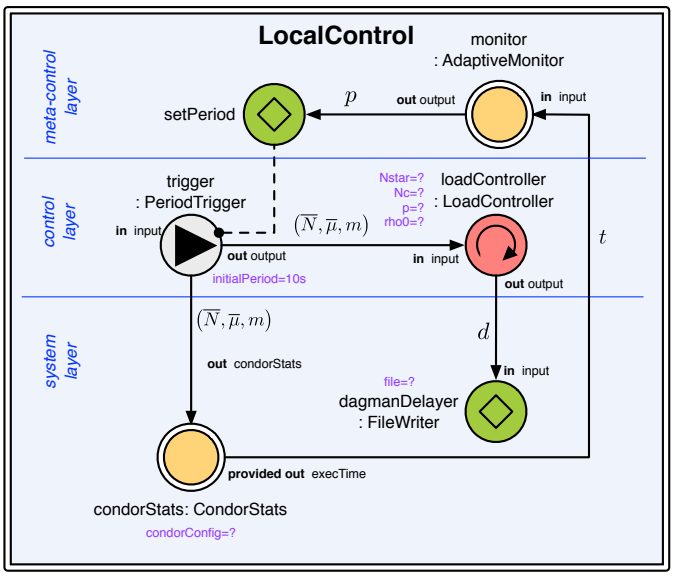


Figure 8.5: `LocalControl` composite



(a) `AdaptiveMonitor` composite



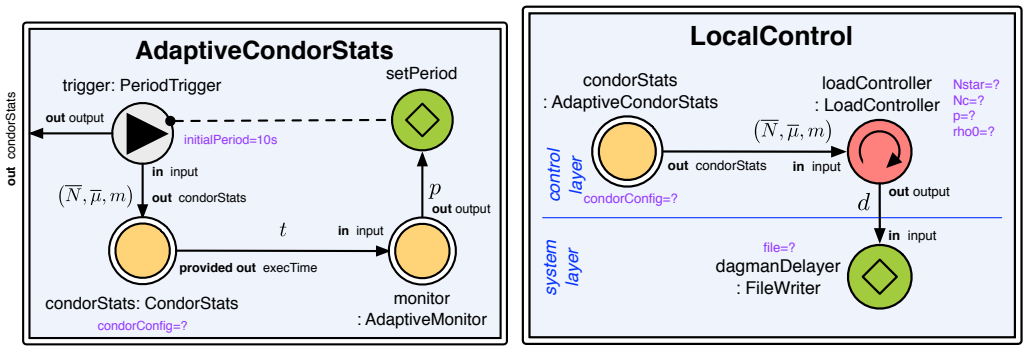
(b) `LocalControl` composite

Figure 8.6: `LocalControl` composite extended with adaptive control

The new architecture is shown in Figure 8.6. We extend `CondorQueueStats` with a provided sensor `execTime` to yield how long it took to execute the `condor_q` command (*cf.*

Figure 8.3). This information is passed to the `PeriodController` that adjusts the monitoring period of the `PeriodicTrigger` proportionally to the execution time of the `condor_q`. We have also added data stabilization to prevent control oscillation and since it represents a generally reusable functionality of an adaptive monitoring, we have additionally extracted it into its own composite, `AdaptiveMonitor`.

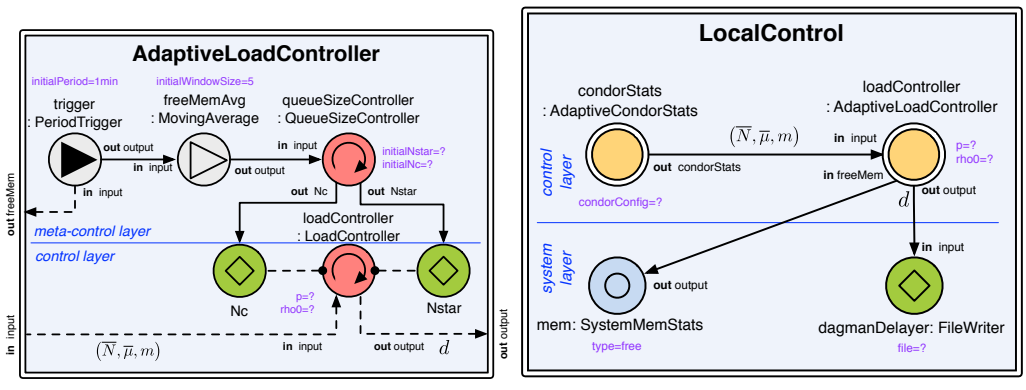
Furthermore, apart from the `LoadController` and `FileWriter`, all the other adaptive elements are part of monitoring that is responsible for computing metrics about HTCondor. We therefore extract them into their own composite `AdaptiveCondorStats`. Figure 8.7 shows the composite and the new organization of the `LocalControl` composite.



(a) `AdaptiveCondorStats` composite (b) `LocalControl` composite

Figure 8.7: Encapsulation of adaptive elements related to system monitoring

Hierarchy of Feedback Control Loops - Adaptive Control So far, the controller tuning parameters such as N and N^* were used as constants, *i.e.*, adaptive element properties (`Nstar`, `Nc`). In Figure 8.8 we show a possible extension of the architecture where both properties are adapted at runtime using adaptive control. First, we create a new composite



(a) `LoadController` composite (b) `LocalControl` composite

Figure 8.8: Encapsulation of monitoring elements

`AdaptiveLoadController` where we place the original `LoadController`. Then we turn the

`Nstar` and `Nc` properties into provided effectors allowing to adjust N and N^* variables. This modification only required to change 4 lines of xFCDL code, from:

```
property Nstar: int
property Nc: int
```

to:

```
provided effector Nstar: int
provided effector Nc: int

act onNstar(Nstar;;)
act onNc(Nc;;)
```

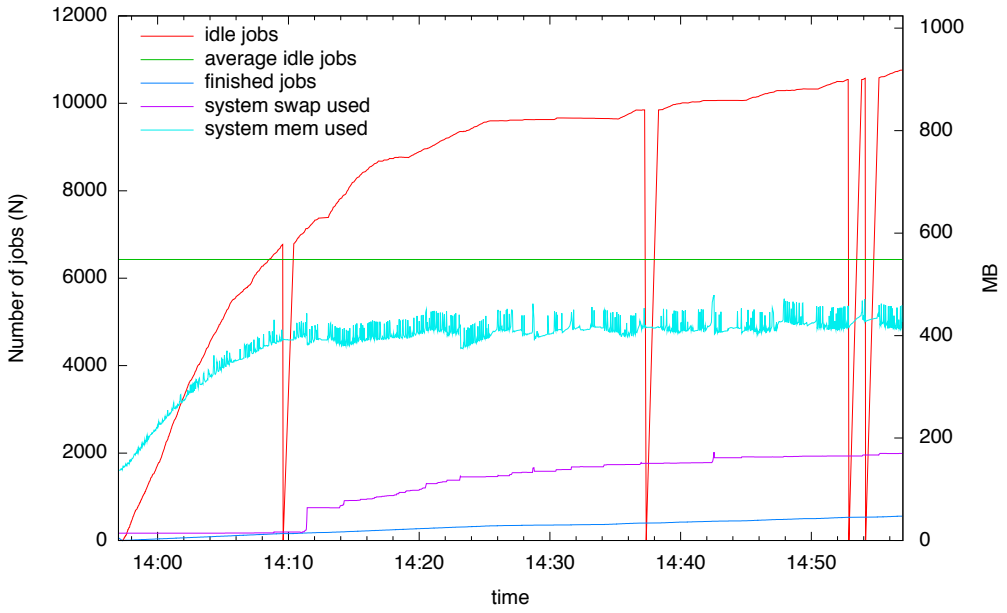
Next, we need to create a new controller and a control model that will appropriately adapt the two parameters. One way is to relate the ideal and maximum queue size to the amount of available memory. Again, we use `PeriodicTrigger` for system observation. It gathers the available memory and passes it through a moving average filter into the `QueueSizeController` that proportionally adjusts the corresponding queue size through the provided effectors. Finally, in the `LocalControl` composite we need to change the type of `loadController` to the newly created composite `AdaptiveLoadController` and connect the `freeMem` port to `SystemMemStats` (change of 5 lines).

The xFCDL code creating the `AdaptiveLoadController` composite has 41 lines including the changes to `LoadController`. On the other hand, it is important to note here, that this scenario extension was made just to demonstrate how an adaptive control can be built using FCDL. Developing a control model that correctly adjusts the N and N^* is not trivial and goes beyond the scope of this thesis.

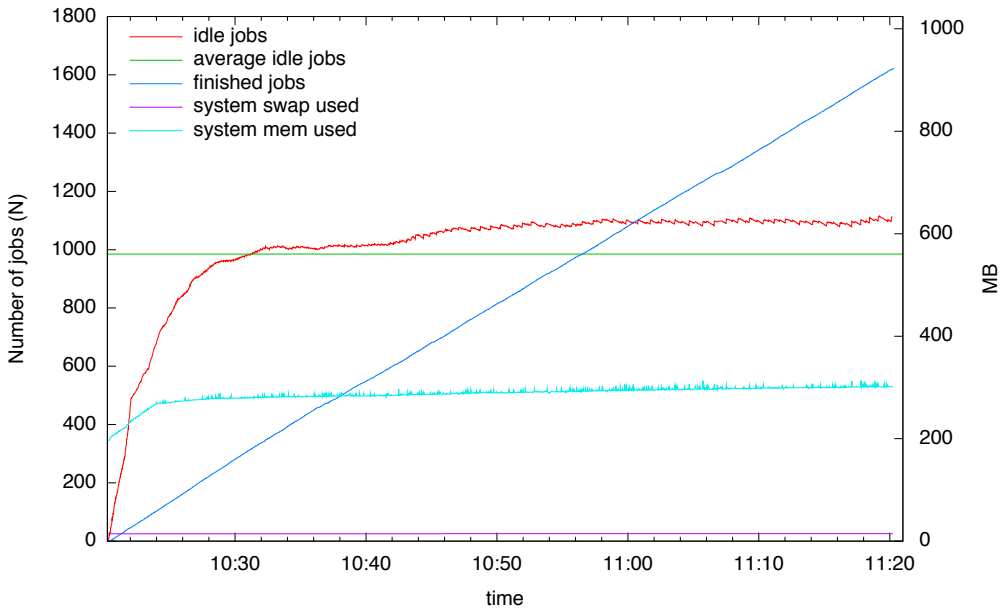
Experimental Results To evaluate the capacity of the resulting architecture (*cf.* Figure 8.7), we set up two HTCondor deployments, with and without the feedback control loops. For both runs we used 20 clients together submitting a diamond-shaped workflow of 1000 tasks, each with a different running time. We initialized the overload control model with[12] $N^* = 1000, N_c = 1500, p = 5, \rho_0 = 10$. Results are show in Figure 8.9 presenting the system behavior without (8.9(a)) and with (8.9(b)) control respectively.

In the feedback controlled system, the amount of idle jobs were for most of the run slightly (6.9%) above the N^*, on average by 69 jobs (116 max). The average load on the host with control was 0.6 in comparison to 3.37 in the system without any control. As shown in Figure 8.9(a), major problems arose when the system started running out of memory and began to swap. After that, it spent most of the time waiting for I/O (on average 51.29% of CPU time were *iowait* comparing to 15.6% in case of the managed system). The gaps in Figure 8.9(a) were caused by timeouts when `condor_q` tried to get the queue information from the schedd agent during the period the system was too stressed (corresponding to load > 9). Finally, one of the important differences is in the amount of work done during the observed period. Because of the resource waste caused by the overload, the unmanaged system executed only 560 tasks while the controlled version did 1620 tasks.

[12] These constants were obtained manually, by profiling the schedd.



(a) uncontrolled



(b) controlled

Figure 8.9: HTCondor local job submission behavior with and without control

Discussion In this case study, we have shown the process of the end-to-end implementation of an overload control in HTCondor. It showed the use of composition to manage the syntax complexities, refining the architecture as new functionalities are being added. We have also showed some reuse, namely the `PeriodicTrigger` processors that were before used in the running scenario (*cf.* Section 3.3.1). Finally, we have briefly presented some experimental results of the system capability.

The experimental setup we use is rather illustrative and does not reflect the industrial scale of contemporary HTCondor deployments [Bradley et al., 2011]. Also, the introduced control model is rather simple. The main purpose of this case study is to demonstrate the capabilities of the proposed approach and the effort of a systematic integration of control mechanisms into a software system and their evolution. In this case, the effort equals to 160 xFCDL source lines of code (SLOC) and 600 SLOC in Java (*cf.* Listing E.1). Interpreting SLOC is always problematic [Jones, 1978], however, what is important about these numbers, for developing a scenario such as the one shown here, is that (1) the 600 SLOC of the touchpoints implementation would have to be implemented in one way or another; (2) the 120 SLOC of xFCDL creates a model and an executable application that can be used directly in connection to a real system, as we have demonstrated in this case study.

The xFCDL code defines the architecture of the system and the implementation of the controllers and Java was used for the HTCondor touchpoints. There are two reasons for using Java for touchpoints implementation: (1) we used Java before to implement HTCondor touchpoints for a similar scenario [Krikava and Collet, 2011] using an earlier version of FCDL and (2) to demonstrate a possible separation of concerns. Control engineers use xFCDL to define the overall architecture and to implement the adaptation engine using Xbase expressions, while the more elaborative and technical processors and target systems touchpoints are developed by programmers and experts in the respective domains using Java, Scala or Xbase within xFCDL. Because of the interaction contracts, there is also no ambiguity of the adaptive elements behavior. Furthermore, thanks to the Eclipse integration both tasks can be realized within the Actress modeling environment (*cf.* Section 5.4) that simplifies and promotes collaboration.

8.1.3 Case Study 2: HTCondor Distributed Job Submission Overload Control

Scenario Overview The previous case study considered only one submission host (one schedd). In this subsection we extend the scenario to cover overload control in a distributed environment. Concretely we experiment with HTCondor-C[13] that allows to move jobs from one submission hosts to another, potentially geographically distributed, providing grid computation mechanisms [Thain et al., 2005].

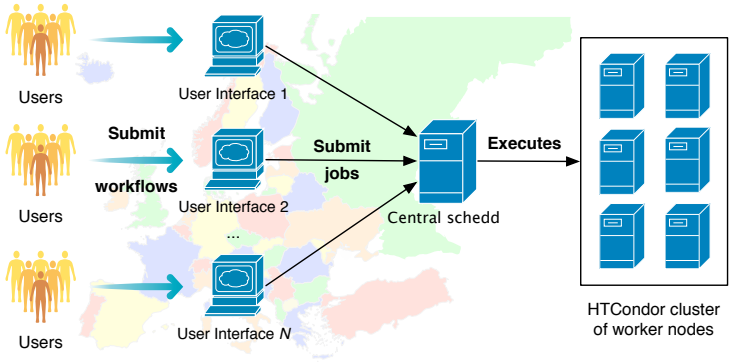


Figure 8.10: Overview of the HTCondor distributed overload control scenario.

Figure 8.10 shows an overview of this control scenario. Each client is an interface for workflow submission similar to the one introduced in the previous section (*cf.* Figure 8.1). The main difference is that this time the local schedulers, running on the user interface clients, do not execute the jobs directly, but rather they move them to the central scheduler.

Adaptation Strategy Similarly to the previous scenario the control is based on adjusting the submission rate according to the current queue utilization. This time however, instead of imposing a delay before submission, we throttle the amount of jobs DAGMans submit at a time. There is a configuration option that constrains how many of the ready to be submitted jobs can be submitted at once (`DAGMAN_MAX_SUBMITS_PER_INTERVAL`). The objective of the control is to dynamically adjust this option based on the current state of the environment.

In order not to overload the HTCondor-C environment, each DAGMan has to consider utilization of two queues: the one of its local schedd, ρ_l, and the central one ρ_c. Using the previously defined queue utilization model (8.1), we can compute the ratio between the ideal queue utilization $\rho(N)$ and the current one:

$$\frac{\rho(N_l)}{\rho_l} \quad \text{and} \quad \frac{\rho(N_c)}{\rho_c}$$

where N_l and N_c denote the number of jobs in the local and central schedd queue respectively. These ratios are used to determine what should be the maximum number of jobs DAGMans submit next. Because we do not want to overload neither of the schedulers, we

[13] http://research.cs.wisc.edu/htcondor/manual/v8.1/5_3Grid_Universe.html

use the smaller value. The maximum number of job submissions, N_{next}, is then:

$$N_{next} = k \left\lceil \texttt{min} \left(\frac{\rho(N_l)}{\rho_l}, \frac{\rho(N_c)}{\rho_c} \right) \right\rceil \tag{8.3}$$

where k is the `DAGMAN_MAX_SUBMITS_PER_INTERVAL` option. One important side effect of this control is that each client can see the indirect effects of the other clients on the schedd. Multiple clients can therefore use one schedd appropriately without being aware of one another.

Feedback Architecture This scenario involves multiple hosts: the local submissions machines (user interfaces), and the central schedd. Figure 8.11 shows the respective architectures in relation to the hosts they run on.

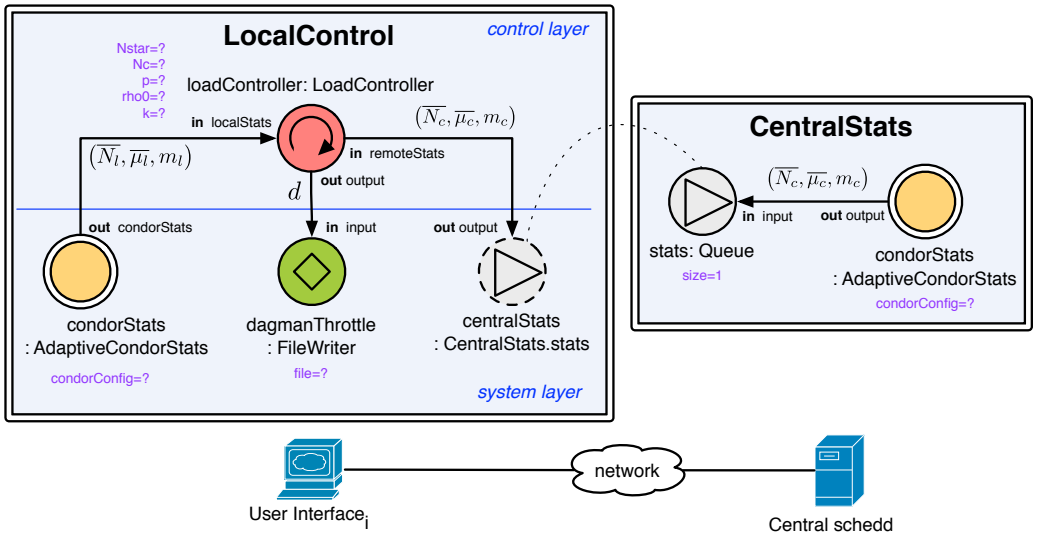


Figure 8.11: `LocalControl` composite for the second case study

- *Central Host.* The system running on the central schedd, represented by the `CentralStats` composite, is only responsible for gathering information for computing the utilization of the central queue, *i.e.*, N_c and μ_c. We reuse the same composite, `AdaptiveCondorStats`, but this time we do not use them for any control, instead, the measured values are only stored in a queue. Notice, that we did not modify the `AdaptiveCondorStats` so the output from this composite also includes the number of running DAGMans, m_c, which shall always be 0. There is a slight overhead of executing the `ProcessCounter` processor, but it is negligible.

- *User Interface.* The adaptation happens at the user interface clients. The responsible composite, `LocalControl`, is very similar to the one we developed in the last section. The only modifications are in the controller. First, since the control considers utilizations of both the local one and the central queue, we extended the controller with an

appropriate input. Second, the controller also implements a different model (8.3). Finally, the new input port `remoteStats` pulls data from the `CentralStats` composite using the referenced feature `centralStats` (*cf.* Section 3.3.6).

Experimental Results To evaluate the architecture we set up an environment using the Grid5000[14] grid computing infrastructure. Grid5000 is a scientific platform allowing reservation of computing resources spread across France for conducting experiments in the domain of distributed computing. The infrastructure allocated for this experiment consists of 10 hosts: one for the central scheduler, one for an execution node allocating 160 execution slots to simulate a pool of HTCondor working nodes, and the rest for user interface clients. In both cases with and without adaptation we ran 64 DAGMan instances in parallel (8 per user interface client), each executing a diagonal-shape workflow of 10000 simple jobs. The same queue utilization model was used for all of the queues, initialized with $N^* = 1000, N_c = 5000, p = 5, \rho_0 = 10$.

Figure 8.12 on page 176 shows the throughput of finished jobs during the first hour of the two executions. Without control the average throughput was less than a half of the one with control, corresponding to less than a half of jobs completed (3950 in comparison to 8090). The run without control also generated about 78% more load at the submission nodes.

Discussion This case study extended the previous with a distribution aspect showing a practical use of remote adaptive elements. It has also demonstrated the possibility of adaptive elements reuse in similar adaptive scenarios. This is one of our objectives, to enable researchers and engineers to experiment with different strategies and mechanisms without investing much effort. The new architecture requires 87 additional lines of xFCDL code and 50 lines of Xbase expressions for the new controller (*cf.* Listing E.2). It reuses all the adaptive elements developed for the local overload control, but the `LoadController`. With the modularity offered by xFCDL, one can start building a library of reusable components that is particular to one's domain.

	Number of *AE* types		Number	SLOC		
	new	reused	of *AE* instances	xFCDL	Xbase	Java
Running Example[1]	12	0	12	169	67	97
Case Study 1	14	1	16	176	40	355
Case Study 2	4	13	125[2]	87	50	0

Figure 8.13: Summary of the number of element and SLOC of the case studies. *AE*: Adaptive Element. [1]Measures based on the version showed in Figure 3.8 corresponding to Listing D.1. [2]14 instances per `LocalControl` per user interface client, 13 instances per central schedd.

[14] http://wwww.grid5000.fr

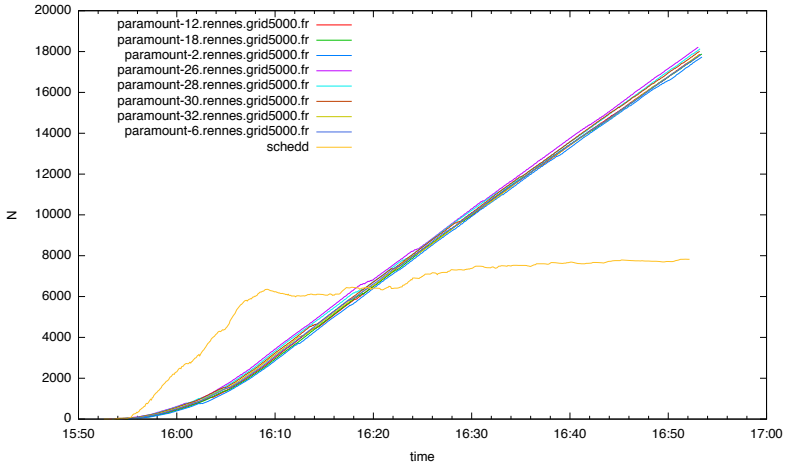
Figure 8.1.3 summarizes the number of elements used in all three case studies. It demonstrates the possibility of reuse of adaptive elements among the scenarios. It also shows that the overall effort in SLOC is rather low. In particular, the xFCDL code related to the system structure is rather systematic once the architecture is settled (*cf.* Listings E.1 and E.2).

Similarly to the previous case study, the experimental setup and the control model we use are rather illustrative. However, it provides a complete infrastructure that can be used by FCL practitioners to seamlessly experiment with more advanced control mechanisms.

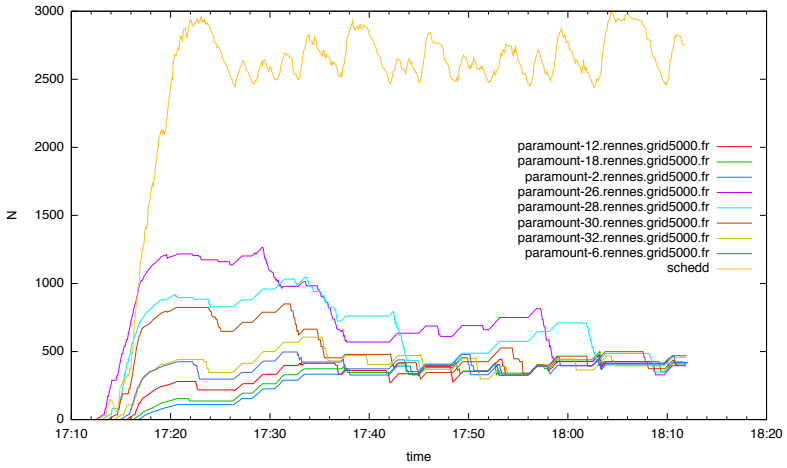
Related Work There are other works experimenting with self-adaptive behavior for DAGMan. For example, Kalayci *et al.* [Kalayci et al., 2010] propose a distributed and adaptive execution of DAGMan workflows by modifying the workflows themselves at runtime in order to be able to submit parts of the workflow into different resource domains. However, this work is trying to solve a different problem, executing workflows as soon as possible in cases more than one Condor pool is accessible.

Acknowledgments The authors would like to thank the HTCondor team from the University of Wisconsin-Madison for all their support.

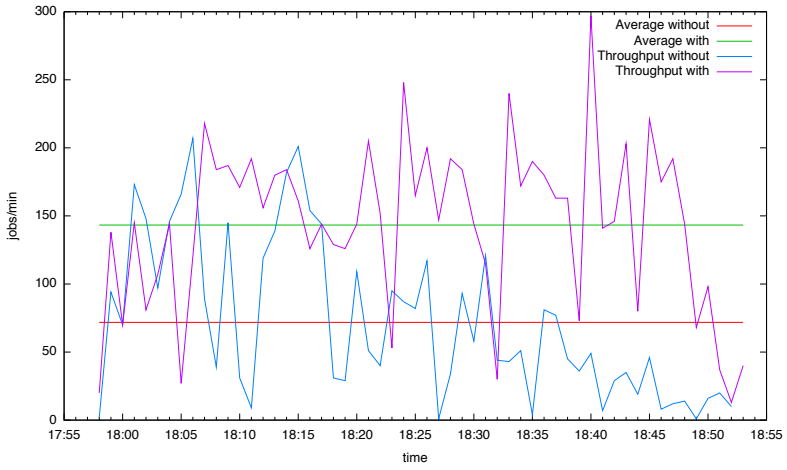
Experiments presented in this section were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies.



(a) uncontrolled



(b) controlled



(c) overall throughput

Figure 8.12: HTCondor distributed job submission behavior with and without control

8.2 Assessing FCDL and the Actress Modeling Environment

In this section we further discuss the advantages and disadvantages of both the FCDL modeling language and the Actress modeling environment and evaluate them against relevant criteria and quality attributes.

The goal of FCDL and the Actress implementation is to integrate self-adaptation mechanisms into software systems. For a long time, evaluation of self-adaptive software systems has been acknowledged to be one of the major concerns [Kephart and Chess, 2003, McKinley et al., 2004, Salehie and Tahvildari, 2005, Weyns et al., 2012]. Part of that is also the evaluation of the tools, frameworks and runtime supports used for their engineering and executions. Since self-adaptive software systems and autonomic computing are both relatively new and open research areas, there is a lack of commonly used evaluation methods and assessment techniques. There are some studies that propose some evaluation criteria and metrics, for example: McCann and Huebscher [McCann and Huebscher, 2004] use QoS, cost, granularity and flexibility, robustness, degree of autonomy and other properties to evaluate autonomic systems. Cheng and Hariri [Chen and Hariri, 2007] present a set of metrics such as scalability, adaptability, overhead, latency, complexity, and effectiveness to evaluate their self-management systems focusing on the quality of the adaptation engine. Neti and Muller *et al.* [Neti and Muller, 2007] developed a framework for an analysis and reasoning about self-healing systems expressing qualities such as support for detecting anomalous system behavior or failure diagnosis. Villegas *et al.* [Villegas et al., 2011] suggest a framework for evaluating quality-driven self-adaptive software systems focusing on the control part of these systems exercising it on 16 approaches.

The primarily focus of these studies is on the self-adaptive mechanisms, assessing the qualities of the control. Except the last one, the evaluation methods are usually exercised only on the particular approaches within the same study that introduced them. An exception is the work of Asadollahi *et al.* [Asadollahi et al., 2009] in which the authors examine the strengths and weaknesses of their StarMX framework (*cf.* Section 2.2.2 on page 25) using a number of criteria synthesized from other studies adjusted to evaluate adaptation enabling frameworks and tools. These criteria express some of the most important capabilities related to self-adaptive characteristics. They are provided to be used as a basis for evaluating different engineering approaches in order to determine their relative properties and enable comparing different solutions to one another. While the proposed set might not be complete and it is rather qualitative, according to us, it gives a reasonable base to start with and we use it as a part of the FCDL and Actress evaluation.

The rest of the discussion in this section is organized as follows. First, we review the application of both FCDL and Actress. Next, we provide a summary of the approach self-adaptive capabilities based on the set of evaluation criteria developed by Asadollahi *et al.* [Asadollahi et al., 2009] and we assess the quality attributes and limitations of our solution. Finally, we discuss how is the approach addressing the rest of the objectives identified in Section 1.2, namely providing higher-level abstraction and development process automation.

8.2.1 Application

The FCDL modeling language has become the core model for the SALTY (*Self-Adaptive very Large disTributed sYstems*) project[15] through which this thesis was funded. The project main aim is to provide an innovative self-managing software stack for runtime self-adaptation of large-scale distributed systems. To realize this objective it relies on the use MDE techniques and principles of autonomic computing to provide end-to-end model-driven approach for building these systems. FCDL contributes to SALTY objectives by proving a flexible higher-level abstraction for defining external self-adaptive software systems through explicit FCLs refined as first class entities.

Within the scope of the SALTY project, another alternative implementation of FCDL has been developed. CORONA (*Control Oriented approach for buildiNg Autonomic systems*) [Nzekwa, 2013] is an approach for engineering amenable autonomic systems based on *Service Component Architecture* (SCA). It uses FCDL[16] for feedback control architecture specifications, but provides its own code generation, verification and runtime support. Instead of using xFCDL[17], it uses Sigma model navigation and modification support (*cf.* Section 7.1) for directly authoring FCDL models in Scala. Resulting FCDL adaptive elements are translated into SCA components and composites using FraSCAti [Seinturier et al., 2009] as its runtime platform targeting self-configuration and self-optimization of component-based systems such as SCA ones. With the notion of components, properties, provided and required services and composites, the SCA model provides a feasible target for FCDL. However, since SCA does not use actor-oriented design, the underlying implementation requires more engineering effort in order to implement the FCDL model of computation and component isolation. Nevertheless it raises our confidence in the MDE approach we have chosen and the ability of FCDL to be used for different runtime platforms. CORONA also builds a set of services on the top of FCDL models such as location optimizer, computing an optimal distribution of adaptive elements for a given network topology and verification heuristics detecting architectural conflicts within FCLs. Since these services work on the same FCDL model, both are usable within the Actress modeling environment.

8.2.2 Self-Adaptive Characteristics and Capabilities

This subsection summarizes the possible self-adaptive characteristics and capabilities of resulting systems built using FCDL and Actress. As discussed at the beginning of this section, we evaluate this using Asadollahi *et al.* [Asadollahi et al., 2009] criteria. Extracting similar metrics and evaluation characteristics from other studies (*cf.* the introduction section), they suggest to use following properties to evaluate the various aspects of frameworks and tools for engineering self-adaptive software systems: *degree of autonomy*, *control scope*, *self-* properties support*, *management logic expression* and *runtime updating management*

[15] An ANR (Agence Nationale de la Recherche) funded research project under the contract ANR-09-SEGI-012: https://salty.unice.fr/

[16] An earlier version.

[17] xFCDL was not available at that point.

logic. Furthermore, in his thesis, Asadollahi [Asadollahi, 2009, page 64] adds 6 more criteria: *control loop construction*, *monitoring technique*, *data communication facility*, *remote management*, *applicable environment* and *managing non-Java systems*.

In this subsection we use these evaluation properties to summarize the self-adaptive capabilities of our approach. In the second set we merge *managing non-Java systems* property into *applicable environment* as in general, we do not find the need to handle Java systems differently. We also rename *management logic expression* to *adaptation logic expression* denoting the mechanisms for defining the self-adaptation mechanisms.

Figure 8.2.2 summarizes the capabilities of both FCDL and Actress. For each property we state the capability of our approach together with an explanation and a reference for more details. We distinguish between FCDL (a general model) and Actress (one possible implementation) where applicable. To improve readability, under the name of each property we also put its definition taken from the Asadollahi's thesis [Asadollahi, 2009, page 64].

The summary of the StarMX framework is given in Asadollahi thesis [Asadollahi, 2009, page 65]. For the other related work (*cf.* Section 2.2.2), we are unable to provide a comparison due to the lack of public access to the frameworks (*cf.* Section 2.2). The exception is Ptolemy, which is publicly available[18] and whose summary of capabilities is shown in Figure 8.2.2.

8.2.3 Quality Attributes

Regarding self-adaptive software systems, Patarin and Makpangou [Babaoglu et al., 2005] mention three software quality attributes for a platform for self-adaptive software system engineering, namely *flexibility*, *performance* and *usability* (*cf.* the citation in Section 2.3). Asadollahi *et al.* [Asadollahi et al., 2009] further adds *scalability*, *reusability* and *extensibility* to cover different perspectives for a software quality evaluation of self-management supporting frameworks. Similar attributes are also discussed and used by Chen and Hariri [Chen and Hariri, 2007]. For evaluating FCDL and Actress we use these quality attributes and further add *testability* that we consider to be of an equal importance.

The rest of this section provides a review of these quality attributes for both FCDL and Actress. Similarly to the others [Asadollahi et al., 2009, Villegas et al., 2013], the assessment is mostly qualitative and as such it tends to be rather subjective. Where possible, we present quantitative results and put a reference to a respective section where the corresponding feature is presented in details.

Flexibility *The ability that allows the developer to combine his own mechanism, algorithm, or technique in the design and implementation of the self-management logic* [Asadollahi et al., 2009].

The feedback control loop in FCDL is decomposed into a number of explicit interconnected adaptive elements, processes similar to functional blocks in block diagrams from control theory (*cf.* Section 2.1.1). The model can represent the main three types of control

[18] http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/index.htm

Criteria	FCDL/ Actress Capabilities
Degree of autonomy The capability of a framework in automating the management process, which ranges from manual to fully autonomic [IBM, 2006].	Closed Loops FCDL focuses on fully automated management processed that is backed by a closed FCL (*cf.* Section 3.3.1).
Control Scope The granularity or the scope of what is being managed [IBM, 2006].	Multiple heterogenous instances FCDL supports multiple FCL that can control various software resources (*cf.* Section 3.3.1).
Self-* property support The capability of the framework in properly addressing different self-* properties.	Potentially any self-* property is supported FCDL is using classical FCL (*cf.* Section 2.1.2) supporting various control schemes and both parametric and structural adaptation (*cf.* Section 3.3.5). Thus it is believed that any self-* properties can be implemented. The case studies presented in this thesis focus on self-optimization but we pointed out that the model was successfully used for self-configuration as well [Nzekwa, 2013]. Nevertheless, more evidence is needed and it rests as a part of our further work (Section 9.2.1).
Adaptation logic expression The mechanisms for defining the self-managing requirements.	Any / Xbase or JVM-based languages FCDL does not pose any restrictions for adaptive element implementation. Actress provides a dedicated support for Xbase and plain Java/Scala (*cf.* Section 5.1.3).
Runtime modification The capability of the framework in allowing runtime modification of the management logic.	Supported FCDL supports reflection (*cf.* Section 3.3.5) allowing for hierarchical and other control schemes to be realized (*e.g.* adaptive monitoring and adaptive control *cf.* Section 8.1.2).
Control loop construction The mechanisms and facilities provided to support creating closed control loops.	Actor-oriented architecture / modeling environment FCDL uses actor-oriented architectural models to represent FCL. Actress facilitates their construction by providing support for modeling, verification and code generation (*cf.* Section 5.4).
Monitoring technique The capability of the framework in supporting different mechanisms for monitoring or activating control loops.	Event-based FCDL defines active adaptive elements that can be bound to any event source including a timer (*cf.* Section 3.3.3).
Data communication facility The capability of the framework in facilitating communication between control loops or autonomic elements.	Mixed push-pull communication FCDL adaptive elements communicate through a mixture of push-pull ports (*cf.* Section 4.1.2).
Remote management The ability of the framework to enable managing a system remotely and transferring the adaptation cost to a different machine.	Embedded FCDL supports remotely distributed adaptive elements through location transparency (*cf.* Section 3.3.6).
Applicable environment The characteristics or specification of the environment and the target systems that the framework can successfully work with.	Any / Any supporting Java Standard Edition FCDL is technologically agnostic, however Actress runtime is JVM-based requiring Java Standard Edition and thus not suitable for embedded systems for instance (*cf.* Section 5.2.2).

Figure 8.14: Summary of FCDL and Actress self-adaptive capabilities

Criteria	Ptolemy Capability
Degree of autonomy	Closed Loops
Control Scope	Multiple heterogenous instances
Self-* property support	Any self-* Ptolemy can model generic feedback control loops.
Adaptation logic expression	Declarative or JVM-based languages Ptolemy uses a hierarchy of composed actors with scoped and well-defined behavior [Liu et al., 2004]. New actors can be added using Java.
Runtime modification	Not supported
Control loop construction	Actor-oriented architecture / modeling environment including simulator
Monitoring technique	Event-based
Data communication facility	Mixed communication Ptolemy support different models of computation.
Remote management	Not supported
Applicable environment	Embedded systems.

Figure 8.15: Summary of Ptolemy 2 self-adaptive capabilities

systems [Patikirikorala et al., 2012]: feedback control system, feed-forward control system, and their combinations. An FCDL process supports multiple input and multiple outputs whose interactions are precisely guided by interaction contracts (*cf.* Section 4.2). Moreover, adaptive elements reflection (*cf.* Section 3.3.5) provides support for designing complex control schemes such as cascaded control, reconfiguring control, hierarchical control, decentralized control and hybrid control [Patikirikorala et al., 2012]. In all three case studies we have employed hierarchical control for the adaptive monitoring part which is the most used control scheme [Patikirikorala et al., 2012]. Furthermore, the embedded support for remote distribution (*cf.* Section 3.3.6) allows one to design control mechanisms that span beyond the boundary of a single host (*e.g.* managing remote systems).

Unlike most frameworks [Ramirez and Cheng, 2010], our model-based approach does not dictate the system architecture neither the use of any specific technology. Actress is based on Java technology and thus requires Java runtime to be present. However, by no means it is limited to adapt only Java systems. As we have shown in all three case studies, it has been used to adapt native applications.

Finally, our approach supports separation of concerns in the sense that the system architecture and control mechanisms can be defined by control engineers while the implementation of the technical/system-level processors or touchpoints can be carried out by software engineers.

Scalability *The capability of a system in properly handling a growing amount of load in a capable manner* [Asadollahi et al., 2009]. In our context we consider the scalability of the FCDL to

handle large models, of the Actress modeling environment to develop large models, and of the Actress runtime to run large models.

FCDL support for composition (*cf.* Section 3.3.4), polymorphic adaptive elements (*cf.* Section 3.3.2) and interaction contracts (Section 4.2) scales to larger models. Throughout the case studies we have employed these techniques and incrementally reified the FCL architectures.

Actress modeling support based on xFCDL allows one to organize large models into multiple files using Java-like namespaces (*cf.* Section 5.1.2). The generated editing support provides Eclipse IDE integration with rich editing support that eases navigation in larger projects and enable developing, debugging and profiling from within the same environment (*cf.* Section 5.4). Actress also includes support for verification to automatically check assumptions about modeled architectures using structural invariants, connectivity and reachability properties (*cf.* Section 5.3).

The Actress runtime is based on the Akka framework that has low footprint and embraces location transparency to scale horizontally across multiple machines (*cf.* Section 5.2.2 and *performance* part below).

Usability *The effort needed to employ (e.g., learn and operate) the software solution in a particular context* [van Vliet, 2008]. Assessing software usability is difficult since it also depends on the preferences and background of its users, which is subjective by definition [Sendall and Kozaczynski, 2003]. In our case this attribute primarily includes usability of the FCDL language and the Actress modeling environment.

The FCDL language syntax is using concepts from control theory and its syntax is close to the block digram (*cf.* Section 3.3.1). It is based on an actor-oriented model with known concepts such as ports and composites. Relying on the actor-model, the system is highly concurrent while allowing for simple adaptive element implementation without the need to protect mutable state (*cf.* Section 3.1.3). It is technologically agnostic and it only focuses on FCL architecture domain-specific concerns, thus hiding information that are not relevant to the design (*cf.* Section 3.1). Moreover, interaction contracts make the architecture both prescriptive and restrictive, guiding implementations of adaptive elements (*cf.* Section 4.2.1).

xFCDL follows known concepts from Java. It allows adaptive element implementation to be directly coded in Xbase or standalone in Java, Scala or other JVM compatible languages. Finally, the Actress modeling environment is integrated in the Eclipse IDE which might simplify adoption for the users already familiar with it.

Reusability *The extent to which a program (or parts thereof) can be reused in other applications* [van Vliet, 2008].

The three case studies already demonstrated some reusability of FCDL adaptive elements and partial feedback control loop (*cf.* Figure 8.6). As expected, target system touchpoints were reused in similar scenarios. The support for polymorphic adaptive elements makes possible to create generic processors such as `PeriodicTrigger` which has

been used in all of the case studies together with the `MovingAverage`. xFCDL furthermore supports higher-order adaptive elements (*cf.* Section 5.1.3) that should also improve reuse.

The amount of reusability of adaptive elements greatly depends on the way they are defined. For example, in both HTCondor case studies we used `ProcessCounter` to count the number of running DAGMans. We implemented it as a single sensor. An alternative implementation would be to decompose it to smaller and more dedicated elements (*cf.* Figure 8.16) relying on the mentioned support of polymorphic and higher-order adaptive elements. The resulting new components implement rather a general functionality (*e.g.*, a

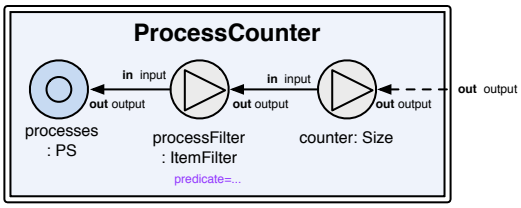


Figure 8.16: A composite implementation of `ProcessCounter`

predicate, a sum) that improves the likelihood of being reused. On the other hand, such a decomposition makes only sense if they will be reused, since the refinement accounts for 50 xFCDL SLOC (three new components with a composite wiring them together). The resulting components are on the other hand easier to test since the functionality is reduced.

Extensibility *The possibility to extend the framework to add new features or to integrate it with other frameworks* [Asadollahi et al., 2009].

FCDL and xFCDL are both defined using their respective EMF meta-models. Therefore, extending their core functionality is only possible by modifying the Actress source code. On the other hand, thanks to MDE, it is possible to use the FCDL models and target different systems, providing new code generators, verification techniques and the like. This has been currently done in the CORONA project [Nzekwa, 2013].

Performance *The amount of computing resources and code required by a program to perform a function* [van Vliet, 2008].

In our case, we consider the amount of the extra overhead caused by execution of the self-adaptive layer. The generated adaptation system is in Java, requiring Java Standard Edition version 1.6 and higher which already involves some memory overhead (in comparison to systems like HTCondor). A single instance of the Actress runtime with no composites deployed accounts for 1.5MB [19]. The Actress domain framework is based on Akka. In Akka 2.0 version, the memory overhead is about 400 bytes per actor instance (2.7 million actors per GB of heap) with a possible throughput of 50 million messages per sec on a single machine[20]. The size of an adaptive element is mostly affected by the amount of

[19] All further measurements were conducted on MacBook Pro 2.53 Ghz Intel i5, 8GB RAM, Java 1.70_17, Akka 2.2.0

[20] `http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single-machine`

state it keeps. The same applies for the execution time whose majority is spent in running the user-code of adaptive element activation methods (throughput of 5000 `PUSH` messages per second amount for 5% of CPU time). The main potential performance issues is in the in-direct load caused by the sensors and effectors. That might be significant (*e.g.*, unbounded execution of `condor_q` in the case of the experimental case studies in Section 8.1) and it must be taken into account while designing any self-adaptive software system.

Testability *The effort required to test a program to ensure that it performs its intended function* [van Vliet, 2008].

FCDL models are amenable to automated consistency checking including user-defined invariants (*cf.* Section 5.3.1) and external model verification of connectivity and data reachability properties via the SPIN model checker (*cf.* Section 5.3.2).

In addition, the behavior of all participating adaptive elements should be tested. For testing adaptive elements implementation, there is no particular support needed, since they can be simply tested in isolation without the Actress runtime. The behavior is contained in *adaptive element act* (*cf.* Section 4.2.8) classes organized into a set of ordinary methods where each corresponds to a particular element behavior. Therefore any of the unit testing frameworks such as JUnit [21] can be used.

8.2.4 Limitations

This subsection discusses the main limitations and shortcomings of both FCDL and Actress.

While FCDL is using static typing and does not enforce any particular data type system, xFCDL relies on the Xbase data type system which is the one of Java. One of the limitations is that it does not support physical units. For example, there is nothing to prevent typing errors such as `speed = time / distance` neither `speed = speed1 + speed2` where `speed1` is in kph and `speed2` in mph. Xbase provides convenient ways for expressing mathematical equations, but it might be too low level for control based on concepts such as decision tables, ruled based policies or state transition diagrams. Languages like Drools [Browne, 2009] can be used through their respective API, however, they are not embedded directly in the adaptive element definition. The Xtext framework we use for xFCDL implementation is rather limited in the ability of composing multiple domain-specific languages [Voelter, 2011]. Both issues are further discussed in Section 9.2.1 and 9.3.1 respectively.

The current support for error handling is using only the default policy which keeps restarting adaptive element certain number of times before it shutdowns. While this policy is configurable in the launcher (Section 5.2.1) it is only modifiable by using the Akka API. Similarly, this is a part of our further work and discussed in Section 9.2.1.

Finally, the external adaptation relies on the fact that the target system is able to provide (or be instrumented to provide) all the required touchpoints. While in general, this

[21] `http://junit.org/`

is not an unreasonable assumption, we have shown that in the case of DAGMan, we had to actually modify its code. On the other hand, there are no easy solution for this next to direct code alteration or AOP techniques, both requiring access to the source code.

8.2.5 Discussion

In this section we complement above evaluation and discuss how is our approach addressing the objectives identified in Section 1.2, namely raising the level of abstraction and automating the development process.

Providing Higher-Level Abstraction As we have shown in the three case studies, using FCDL developers work on a higher-level of abstraction using concepts from the self-adaptive system domain. Without a domain-specific modeling language like FCDL, developers would have to use GPL such as C++, Java or Scala that do not convey domain-specific concerns and semantics [Schmidt, 2006]. The main advantage of a higher abstraction level is that it narrows the gap between the problem domain and the implementation domain that in turns helps to reduce the accidental complexities. In FCDL, an FCL is described in a model by using directly the problem domain terminology and the implementation of the system is synthesized from such a description (*cf.* Section 5.2.1).

This higher-level abstraction is not only reflected in the overall architecture, but also in the implementation of adaptive elements. In particular, all the semantic rules we have developed in Chapter 4 are embedded in the underlying domain framework allowing developers to focus solely on the implementation of the activation methods (*cf.* Section 5.2.2). Figure 8.17 shows the different abstraction levels and the corresponding code (or code excerpt) needed for implementing the `PeriodicTrigger` processor.

It is important to note here that the abstraction we have chosen for describing feedback control is not the only one and it is possible to have even higher-level models than FCDL. The advantage of FCDL is that it matches block diagrams that provide an established abstraction of feedback control loops (*cf.* Section 2.1.1). It is flexible, yet rigid enough for automated synthesis of a complete control system. The abstraction levels are further discussed in Section 9.3.1.

Development Process Automation Chapter 5 introduced the Actress modeling environment. In particular, in the last section we have shown how the modeling, code generation and verification supports are integrated into the Eclipse IDE providing and integrated development experience. Using Xbase for implementation, the code generator emits a complete executable applications, yet with customization and configuration opportunities. The custom implementation of adaptive elements further separates concerns between control mechanisms and infrastructure development. Moreover, the generated skeleton implementation based on interaction contracts is both prescriptive and restrictive, guiding developers in the sense of what the architecture allows (*cf.* Section 4.2.8). Next to the code generator, the verification support automatizes model consistency checking including user-defined structural constraints.

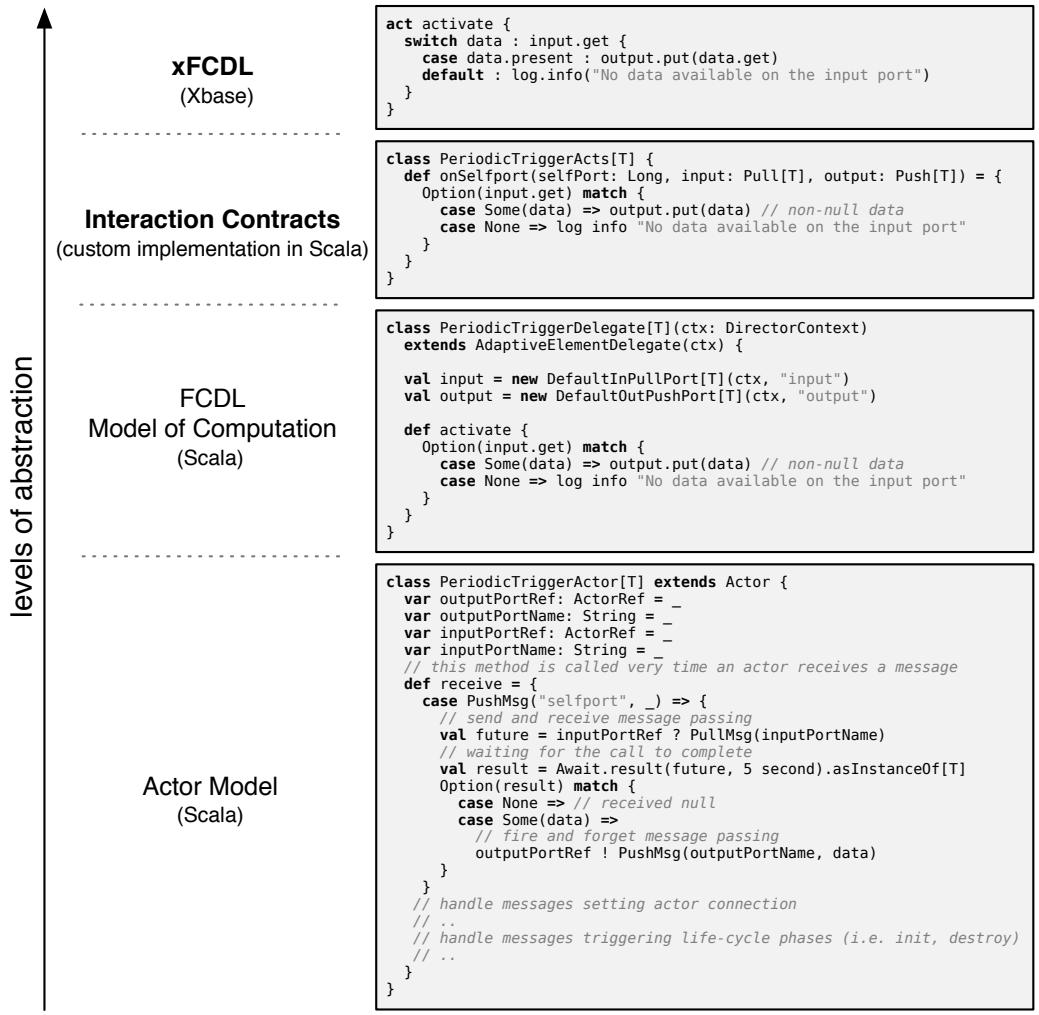


Figure 8.17: Different levels of abstraction in Actress system. The emphasized levels are the ones by FCDL and Actress

Moreover, during the implementation of the case studies, we observed, that the automation of the development process helps developing the solution incrementally. As we have shown in Section 8.1.2, we have effectively started with a basic control scheme and refined it step-by-step into a more advanced one. At the end of each step Actress generated the complete code that could be directly tested and executed. This notably seems to allow for continuous delivery, one of the pillar of the *agile development process* [Martin, 2003].

8.3 Assessing Sigma Model Manipulation Languages

In this section we discuss some of the benefits and limitations of the Sigma model manipulations DSLs. The discussion is organized similarly to the previous section. First, we review the application of Sigma. Next, we assess the quality attributes and discuss Sigma

limitations and the *lightweight*-ness of the approach (one of the objectives identified in Section 1.2). Finally, we briefly review other internal DSL-based approaches to model manipulation.

8.3.1 Application

Sigma has been mainly employed in the context of the SALTY project. It has been used to implement the Actress modeling environment. Also, the previous mentioned CORONA project [Nzekwa, 2013] has used the model navigation and modification support for authoring FCDL models. Finally, the Yourcast[22] project has also adopted Sigma for model delegation and M2T transformations replacing Velocity[23] and plain Java templates.

The main motivation for Sigma were the shortcomings of OCL used for the initial Actress implementation [Krikava and Collet, 2012a]. As we have already mentioned in Section 5.3.1, we use extensively model consistency checking (having over 140 structural constraints) and in particular the missing support for invariant guards was causing a lot of code duplication as well as hard to write and read expressions. Another advantage of using Sigma was the possibility to add helper methods (reusable across model manipulation tasks) without the need of cluttering the meta-model. We used both the imperative and hybrid style of M2M transformations. We found that the imperative style is more suitable in the case one has to interface with Xtext during the JVM model inference. It has allowed us to develop the transformation code faster, especially considering the evolution of the meta-models during the project. For the xFCDL to FCDL transformation as well as for the FCDL types instantiation we have used the hybrid style since these meta-models are close to one another. The code-explicit style of the M2T transformation is particularly suitable for code generation. The provided decorators (*e.g.*, smart whitespace handling, relaxed newlines and indents) helps keeping the readability of the template by following the target text layout. The M2T transformation is also used in Sigma itself for generating the common infrastructure (*cf.* Section 7.1).

In the case of YourCast, the use of Sigma M2T transformation resulted in 20% code size reduction. Primarily, this is due to more expressive model navigation and more compact text outputting constructs.

8.3.2 Quality Attributes

We organize the review about Sigma quality attributes in the similar fashion as we did in Section 6.3 where the issues of the selected model manipulation languages were discussed. However, instead of versatility we discuss usability since an internal DSL inherits the versatility of its host language, but can have potentially issues with the domain-specific constructs.

Expressiveness As for expressiveness regarding model navigation and modification, it is almost identical to the one of OCL or EOL (*cf.* Section 7.1). Unlike EOL, the languages

[22] http://www.yourcast.fr/
[23] http://velocity.apache.org/

are fully[24] type safe and therefore the runtime typing errors are obviously reduced. Moreover, in some cases the use of pattern matching and extractor can further improve the readability, particularly in expressions dealing with type checking (relieving from *type-ifs* such as `instanceof`, `oclIsKindOf` and the like). The model consistency checking DSL implements all the EVL features including support for invariant reuse (Section 7.2). Regarding the model transformation DSLs, they include similar features that are found in their respective external DSL counterparts. However, since some concepts are expressed differently (*e.g.* transformation rules inheritance), more experiments will have to be conducted for a fine-grained assessment.

Usability As we have already mentioned, the usability aspect is a complicated subject since it largely depends on the preference and background of its users [Sendall and Kozaczynski, 2003]. Considering basic set of Scala skills (more on this is discussed in Section 8.3.3), the provided API is rather small and thus less effort is likely to be required in comparison with learning new language such as OCL or EOL. All DSLs support common features presented in the other languages such as *modularity* to allow organizing task-specific concerns into modules, their *reuse* without the need to duplicate them, and *extension* of meta-models to access auxiliary information that are not or should not be presented in the meta-model (*i.e.* helper methods and queries).

Furthermore, Sigma offers the possibility to implement derived properties and operation bodies directly in Scala (or Java) in a more sustainable way, by relying on the generation gap pattern (*cf.* Section 7.1.4).

Executing a model manipulation task in Sigma is no different from executing a regular JVM-based application. There is no interpreter, no particular execution environment required which simplifies its integration into existing software project. It can be included into any build tool since it does not require any extra treatment besides compilation.

While it primarily targets Ecore models, it is likely to be usable in different contexts and meta-modeling environments. For example the model-to-text transformation can be used as a general engine library for producing text outputs.

Testability The method-based styles of all the three languages allows to cherry-pick the fragments of model manipulation to be tested by any Java-based unit testing framework. This is especially useful for larger model transformations. For example, the following is an excerpt of a test of the adaptive element declaration to adaptive element type transformation rule from Listing 7.3.2 using ScalaTest[25] unit testing framework:

```
1 test("AdaptiveElementDecl to AdaptiveElementType rule") {
2   // create the source elements
3   val source = AdaptiveElementDecl(name = "anElem", active = true)
4   souce.ports += PortDecl(name = "aPort", portType = INPUT, portMode = PUSH)
5   // create an empty target
6   val target = AdaptiveElementType()
7
```

[24] It is still possible to make wrong explicit cast using the `asInstanceOf` method.
[25] http://www.scalatest.org/

```
8   // execute transformation
9   new OO2DB().ruleAdaptiveElement(source,target)
10
11  // verification
12  target.name should be ("anElem")
13  target.ports should have size 1
14  inside (target.ports(0)) { case Buffer(Port(name, portType, portMode)) =>
15    name should be ("aPort")
16    portType should be (INPUT)
17    portMode should be (PUSH)
18  }
19 }
```

The Sigma support for model modification allows one to quickly prepare snippets of model instances to test the transformation rules (lines 2-6). On line 9 we cherry-pick the rule to test. Finally, lines 12-18 shows one possibility how to assert the assumptions about the rule behavior.

Interoperability, Consistency and Reuse Since all the DSLs are build on top of the same host language, they are interoperable. Not only can they be organized in a workflow by simply calling appropriate methods without introducing yet a new orchestration language, but it is also possible to embed them together. For example a M2T transformation into a M2M transformation rule. We rely on this in Actress where we use M2T transformations to synthesize method source code during the xFCDL to JVM transformations. There is also a possibility to interact with the other model manipulation languages though their respective Java API.

We have shown that the DSLs are constructed though the same pattern, organizing the task-specific concerns into classes and methods making them consistent.

An existing model manipulation tasks can be reused by either importing or extending the corresponding model classes. Furthermore, the Scala mixin-class composition is used to modularize cross-cutting concerns particularly for the model transformations. Finally, we rely on Scala explicit conversion to extend existing meta-models with auxiliary operations and properties.

Tool Support One of the advantages of an internal DSL is that it can directly reuse the tool support provided for the host language. The recent versions of the Scala IDE[26] provides solid tools facilitating Scala development including rich editing and debugging support. Moreover, additional tools operating on the JVM class level such as profilers can be directly used. On the other hand there are also some shortcomings associated to this that are discussed in the next section.

Furthermore, while Sigma has a strong dependency on EMF, it does not require the Eclipse environment. Consequently, the development of EMF based applications, which were traditionally tied to Eclipse IDE, can be carried out in other development environments.

[26]Scala Eclipse integration http://scala-ide.org/

Performance The performance of Sigma model manipulation tasks is determined by two factors: the performance of the host language, *i.e.*, the quality of Scala compiled bytecode, and the overhead of the Sigma API. As a part of the evaluation, we have implemented the same M2T transformation in Sigma and all the concerned M2T languages[27]. In addition, we have also implemented it in pure Java and Scala with no additional libraries. We use the Java version as a performance baseline and the pure Scala version to measure the overhead of Scala in comparison to Java and the performance penalty caused by the Sigma API in comparison to pure Scala. The implementation in the other languages is there to evaluate our requirement for Sigma to perform similarly as the other approaches. We choose to use the M2T transformation since (1) it uses the most Scala features such as implicit conversions and string interpolation, (2) M2T is after model consistency checking[28] used the most in Sigma, and (3) the implementation in the other languages was straightforward, limiting the possibility of misusing some features. As a concrete M2T transformation scenario we choose a sample UML-like model into Java classes transformation since nearly all of the listed languages provide an example that is based on it. The result of a median of 20 consecutive runs for two different model sizes is shown in Figure 8.18.

# of C,P,M	Java	Sigma	EGL	Acceleo	Xtend	Kermeta	Scala
250,50,50	1.0	1.03	18.59	11.88	0.94	0.97	0.94
500,100,100	1.0	1.89	48.14	16.43	1.00	1.00	0.96

Figure 8.18: Sample performance of different M2T languages. C,P,M: the size of the model as a number of classes (C) each having (P) properties and (M) methods.

As expected, the performance Sigma together with the compiled languages is close to the one of Java while the interpreted approaches are an order of magnitude apart (also their memory footprint is double). The decrease in Sigma performance in the case of the larger model is caused by the whitespace handling decorator. Every appended string is checked whether there are whitespaces to remove and its complexity increases with the indent level. Without this decorator, the performance is again close to Java (0.91 in the first case, 0.99 in the second case).

8.3.3 Limitations

Obviously there are also some shortcomings in Sigma, mostly related to the limitations of the internal DSL approach. Apart from the syntax restrictions, an internal DSL is in general a leaky abstraction [Siek, 2010]. This can be exploited in multiple ways. For example an implementation of guards and structural constraints can contain arbitrary code. By default, there is no simple way to make sure that the expressions are side-effect free

[27] Kermeta version was put together by Didier Vojtisek, a Kermeta committer. All the source code is available from `https://github.com/fikovnik/Sigma/tree/develop/examples/fr.unice.i3s.sigma.examples.performance`

[28] The problem with evaluating model consistency checking across different languages is that they do not provide a standardized interface and they capture different amount of details.

without employing an external checker such as IGJ [Zibin et al., 2007]. Adding such a checker again increases the complexity of the overall project setup.

Despite the advantages of Scala Eclipse tools, the main problem is that these tools currently do not have any understanding of the domain-specific aspect of an internal DSL. While we were able to address some issues related to compiler error reporting (*cf.* Section 7.3.2), there is currently no similar support for debugging. For example, debugging a M2M transformation simply exploits the full stack frame instead of showing the chain of different transformation rule applications. This issue has been already well identified and there are some prospects in using Scala language virtualization [Moors et al., 2012].

Traditionally, the support for domain-specific analysis and error checking has been difficult to realize in internal DSLs. However, Scala offers some more advanced methods for *deep* DSL embedding, using language virtualization and lightweight modular staging [Rompf and Odersky, 2010]. Nevertheless, these two problems also appear in the external DSLs that provide imperative constructs and the black-box interoperability with Java.

Similarly is the support for domain-specific optimizations difficult to realize in internal DSLs. Such an example is the performance issue identified above. With Scala macros and lightweight modular staging we should be able to alleviate such an issue in the way that the decorator is applied only once at the compile time, rather than for each and every append at runtime (*cf.* Section 9.3.2).

There are also some aspects of the DSLs that are not yet completely type safe. For example, there is currently no compile time checking to ensure that all the first parameters of M2M transformation rules belongs to the same source model. While such a check is rather trivial to realize in an external DSLs, it is very difficult to do in an internal one and it involves complex generic programming with type classes [Oliveira et al., 2010] (again resulting in cryptic error messages).

Besides, depending on the target audience, the use of Scala can be seen as a drawback rather than a merit. It is a relatively new language that has not yet reached the popularity of some of the mainstream programming languages. It might be hard to justify learning a language such as Scala solely for the purpose of model manipulation. We observed that Sigma only requires a basic set of Scala skills, similarly to what would be required from a developer to know in Java in order to use the EMF API. On the other hand, some of the error messages might be rather difficult to digest for a Scala newcomer.

Finally, there is an extra overhead of generating the common infrastructure. This overhead might be partially removed in the next version of Scala, which should support type macros.

8.3.4 Discussion

Throughout the above subsection we have discussed how does Sigma address the objectives identified in Section 1.2. In this last subsection we summarize the main Sigma properties for which we consider it to be a *lightweight* approach to model manipulation as opposed to the more heavy-weight external DSLs: (1) It only relies on Scala with no other

dependencies (external DSLs have usually large dependency stack, *e.g.*, Acceleo requires additional 10 Eclipse plug-ins). (2) It is implemented on a library level, requiring only to get familiar with an API rather than a completely new language. (3) It has small API, relying on existing Scala concepts and functionalities (*e.g.*. organizing task-specific concerns into classes and methods). (4) It can be easily integrated in any build system that supports Scala, not requiring specialized plug-ins such as in the case of external DSLs. (5) Task-specific DSLs can be embedded together, which is currently not supported by the external DSLs. (6) It does not require any special runtime environment, as opposed to the interpreted DSLs. (7) Its performance is close to the one of Java. (8) It is potentially testable with any Java unit testing framework.

Furthermore, being an internal DSL is notably reflected in the code size of the implementation. Sigma is currently implemented in 3500 lines of Scala code. This is an order of magnitude less than EOL or Kermeta, which is an order of magnitude less than Eclipse OCL.

Related Work We conclude the evaluation of Sigma with an overview of some other model manipulation approaches that also target internal DSLs.

Cuadrado *et al.* [Sánchez Cuadrado et al., 2012] made an interesting comparison between the effort of building a rule-based transformation language both as an internal DSL in Ruby and a standalone external one. They concluded that a success of internal DSL highly depends on the selection of the host language and its support for DSL embedding, performance, tool support and popularity, *i.e.*, whether ``*the target audience knows the host language or, at least, is not reluctant to learn it.*'' George *et al.* [George et al., 2012] used Scala to build a model-to-model transformation DSL for the EMF platform that resembles ATL. Since we use the same host language, their DSL is fully interoperable with ours, *e.g.* the common infrastructure (Section 7.1) can be directly used in the transformation rules. Their internal DSL is not completely type safe and they represent transformation rules directly as anonymous classes, which limits their modularity and reusability. Wider [Wider, 2011] presents an interesting approach to bidirectional model transformations by embedding lenses (a combinator-based approach to bidirectional term-transformations) into Scala and showed how they can be used in an MDE context. Akehurst *et al.* [Akehurst et al., 2006] developed a Java library for simple imperative M2M transformations. Being based on Java gives it performance and tool support advantages as well as a wider audience. On the other hand, there is no particular support for improving the expressiveness of model navigation and modification resulting in rather verbose and complicated code (*cf.* Section 6.1).

8.4 Summary

This chapter has presented an evaluation of this thesis. It showed suitability of FCDL of Actress using two additional case studies taken from the HTC domain. They were fol-

lowed by a discussion about the approach application, capabilities and quality attributes. Similarly, we assessed Sigma model manipulations DSLs, presenting its use aside of Actress, quality attributes and limitations. We have also discussed why we consider it to be a lightweight approach to model manipulation and finally reviewed some other work experimenting with internal DSLs in the same context.

The following chapter concludes this thesis and outlines some perspectives for future research.

CHAPTER 9

Conclusions and Perspectives

This final chapter brings this thesis to a conclusion. We summarize our research and outline further work to improve the capabilities of FCDL, Actress and Sigma. We also discuss further research directions expanding our own understanding of MDE application to the domain of self-adaptive software systems engineering.

9.1 Contributions Summary

The drastic expansion of contemporary software systems and IT infrastructures points to an inevitable need for new operation modes to be implemented in order to manage evolving requirements, growing complexity and operation costs. To cope with these issues software systems must become adaptive.

This thesis contributes to this aim by combining the self-adaptive software systems with the principles of model-driven engineering in order to provide a systematic tooled approach for integrating control mechanisms into software applications. This has led to two distinct contributions: the FCDL domain-specific modeling language with the Actress modeling environment and the Sigma family of model manipulation DSLs.

9.1.1 FCDL and the Actress Modeling Environment

Starting with the literature overview we studied feedback control loops and their application into software systems as the fundamental means to achieve self-adaptive capabilities. From the literature study we observed that while there is a number of approaches that focus on enabling adaptation in software systems or on designing control mechanisms, less attention is paid to the integration aspect, *i.e.*, forming the architecture connection between the two of them. This has motivated our objective of providing researchers and engineers with an approach that eases experimentation and implementation of self-adaptation without the need of coping with low-level implementation and infrastructure details.

We proposed a domain-specific modeling language called FCDL for integrating adaptation mechanisms into software systems through external feedback control loops. The key advantage in the domain-specific modeling approach is the possibility to raise the level of abstraction on which the FCLs, their processes and interactions are described, making them amenable to automated analysis and implementation code synthesis. FCDL defines feedback architectures as hierarchically organized networks of adaptive elements. Adaptive elements are actor-like entities that represent the corresponding FCL processes such as monitoring, decision-making and reconfiguration. They interact with one another through a mixture of explicit push-pull communication channels. The model is statically typed, handles composition, supports element distribution via location transparency and is reflective thereby enabling coordination of multiple control loops using different control schemes. It is adaptation domain and technology agnostic making it applicable to a wide range of software systems and adaptation properties.

The language semantics is described using a model of computation that defines operational rules governing interactions among the feedback processes. The model of computation is further complemented with a notion of interaction contracts. They constrain the allowed interactions using formal descriptions enabling various verifications such as architecture consistency, determinacy and completeness.

To facilitate the development using FCDL, a modeling environment called Actress has been implemented. Integrated in the Eclipse IDE, it provides a reference implementation of FCDL together with dedicated support for modeling, code generation and verification. The modeling support is based on a textual domain-specific language, xFCDL, that apart from covering all FCDL concepts, enables modularization and adaptive element implementations through Java-like expressions. The Actress code generator transforms FCDL architectures into executable Java applications, providing a strong mapping between the control system design and its runtime implementation. Finally, the verification support automates consistency checking of FCDL structural constraints including user-defined invariants, as well as connectivity and data reachability properties through the means of external verification.

Throughout the thesis, the approach was illustrated on a concrete real-world adaptation scenario motivated by the work of Abdelzaher *et al.* [Abdelzaher and Bhatti, 1999, Abdelzaher et al., 2002] on web servers QoS management control. To further demonstrate the FCDL and Actress suitability, we have presented an end-to-end implementation of two additional real-world case studies dealing with overload control in contemporary high-throughput computing environments. These applications provided some further evidence of the following benefits: (1) an explicit and concrete representation of FCLs without imposing any particular architecture is provided; (2) each adaptive element is an independent unit that can be developed and tested in isolation, and is reusable in different adaptation scenarios; (3) the reflection capabilities make complex control schemes explicit, allowing multiple coordinated and hierarchically organized FCLs expressed in a uniform way; (4) separation of concerns is obtained between system infrastructure and control mechanisms development; (5) embedded support for remoting simplifies FCL de-

velopment in distributed systems is provided, and (6) incremental development in agile settings with a continuous delivery is possible using the Actress modeling environment.

9.1.2 Sigma Model Manipulation DSLs

The Actress modeling environment relies on FCDL and xFCDL model manipulation for its modeling, code generation and verification supports. We have considered 9 model manipulation approaches for Actress implementation. However, exercising them in the context of Actress highlighted some shortcomings (*e.g.* scalability issues in terms of flexibility and performance of both DSLs and their tool support) in the languages and their provided tool support, significantly contributing to the accidental complexities we experienced. Therefore, we explored an alternative internal DSL approach whereby the specific model manipulation constructs are embedded into a GPL.

Identifying the main principles for internal model manipulation DSLs and a set of requirements for the host language, we chose Scala to develop a family of DSLs for model consistency checking, model-to-model and model-to-text transformations. The DSLs are based on a common infrastructure that bridges the Eclipse Modeling Framework to Scala, providing a set of operations for model navigation, modification and delegation. This makes the DSLs interoperable and consistent. They provide similar expressiveness and features that are found in the external model manipulation DSLs. In addition, they deliver good performance, compact implementation and the ability to take advantage of the Scala tool support with a significantly reduced engineering effort in comparison to the one of external DSLs.

These properties make Sigma a lightweight alternative to the external model manipulation languages in Scala / EMF based projects.

9.2 Further Work

In this section we focus on the short-term to mid-term perspectives that are arising from the current state of affairs of both aspects of our work. They primarily address some of the limitations identified in the respective sections 8.2.4 and 8.3.3. However, among the limitations, there are issues that require further longer-term research to fully comprehend their ramifications. They will be further discussed in Section 9.3.

9.2.1 FCDL and Actress Improvements

Experiments Putting a software product into practical use is the ideal approach to understand its advantages, but more importantly, its limitations and shortcomings. In our particular interest, coming from the SALTY project, which has funded this thesis and focuses on large-scale distributed systems, is to subject it to other case studies in different distributed computing infrastructures. Currently, we are exploring the use of our approach for a self-healing scenario dealing with worlflow activity incidents presented by da Silva *et al.* [Ferreira da Silva et al., 2012]. Another case study we plan to implement

is the *Znn.com* benchmark [Cheng et al., 2009b] as discussed in Section 8.1.1. However, both scenarios concern the same class of systems. In order to get a better picture of the approach capabilities, it is very important to look for additional case studies from different domains focusing on different *self-** properties. For example, a good starting point might be the work of Gaudin *et al.* [Gaudin et al., 2011] and Canzanese *et al.* [Canzanese et al., 2011] in the areas of self-healing of un-handled runtime exceptions and self-protecting against malware infection respectively. Furthermore, apart from identifying pros and cons, work on different case studies should foster a repository of reusable adaptive elements that shall further ease the adoption of FCDL and Actress.

We also intend to formalize the HTC case studies presented in this thesis and offer them to the community.

Design-time Support One of the aims of the design-time support is to catch potential issues before they appear at runtime. Static typing contributes to this objective, however, as we have shown in Section 8.2.4, there are still typing errors that can occur and that can be particularly difficult to debug. One way to address this problem is to further extend data types with physical units. It should be possible for developers to specify units next to data types definitions together with conversion mechanisms, eliminating problems such as `speed = time / distance`. A good starting point is provided by the work of Voelter *et al.* on *mbeddr* [Voelter et al., 2012], an extended C language with additional constructs including physical units.

The capability of reuse in the FCDL model can be further improved by integrating inheritance mechanisms. The issues of inheritance in actor-oriented design has been already studied by Lee *et al.* [Lee et al., 2009], proposing a formal structure with constraints allowing for disciplined form of multiple inheritance with unambiguous inheritance and overriding behavior. We plan to build on that and adapt the mechanisms to support adaptive elements subclassing, inheritance, and overriding.

There are many additional features that would improve the usability and productivity of the Actress modeling environment. For example, it should provide refactoring capabilities such as refining an adaptive element into a composite which was frequently used operation during the implementation of all the case studies. Another missing feature is FCDL model visualization using the graphical notation introduced in Section 3.3, *e.g.*, by transforming the model into a Graphviz[1] model, which is directly presentable in Eclipse IDE[2].

Finally, a deployment of the Actress synthesized system might be an issue, especially in the case of distributed systems and as such it should be addressed by Actress. For this we hope to reuse the work that has been done in the CORONA project [Nzekwa, 2013], but making it tailored for the Actress runtime.

Runtime Support For the adaptation scenarios that need to integrate with native applications (touchpoints interfacing with native libraries) or where the runtime overhead is

[1] http://www.graphviz.org/
[2] Using the Eclipse visualization toolkit Zest http://www.eclipse.org/gef/zest/

an issue, it would be useful to have a native implementation of the Actress runtime in C or C++. In Section 5.2.1 we have already mentioned a possibility of using a different technological stacks for the runtime platform, concretely C++ with libcppa for the actor framework. The main engineering issue lies in developing a compiler that translates Xbase code into the chosen native language[3]. Another opportunity is to allow touchpoints to be integrated directly in the source of the target systems using AOP techniques. In such a case, the target application will contain an embedded version of the runtime, only running the touchpoints, enabling the application adaption. Finally, with multiple runtime platforms it is desirable to make them interoperable and composable, allowing FCL architectures to use a mixture of these different runtimes.

Despite model consistency checking and verification, there is always a possibility for errors occuring at runtime. Therefore, error handling and fault tolerance represents another important area of research. The hierarchical organization establishes the parent (composite) - child (adaptive element) relationship in which the composites are responsible to respond to the nested elements failures. What is missing is a flexible way for the developers to describe the mechanisms that should be implemented in case of failure. They should also be built as feedback control loops, but concerning the control layer. With these mechanisms in place, we can push the user-defined invariants to runtime, *i.e.*, constraining ports value range through pre and post conditions within interaction contracts.

The Actress runtime is reflective and dynamically reconfigurable. As such it should also provide some introspection and management interfaces for developers and system administrators to get more insight about its operation. For example, an Actress console could federate the monitoring and management capabilities of the runtime into some global view in a web-application. The main concerns here are what information are useful to collect and what is their associated cost. Finally, the console could also be used to involve users into the decision making process, *e.g.* to prevent certain adaptation actions from happening. While it might seem to be a counter intuitive to do since it goes against the *self* prefix in the self-adaptive software systems, it can help to improve confidence and the overall trusts in these systems, which in turn may increase their adoption [Cheng, 2008].

9.2.2 Sigma Improvements

Similarly, Sigma has to be exercised more thoroughly in order to better understand the possibilities of the internal DSL approach for model manipulations. Part of our current work in progress consists in carrying out more evaluations to further assess the approach. This should also help us exploring what other advantages of the internal DSLs, beyond the traditional model manipulation concepts, may benefit users.

Next to this, the main focus is to make the languages type safe and alleviate the issues presented in Section 8.3.3. Sigma should also be better integrated with the Eclipse IDE. For example, it should automatically regenerate the common infrastructure every time

[3]The latest C++ 11 standard supports lambda expressions which should significantly simplify the implementation.

a model is changed, a task that currently has to be done manually. However, the major issues identified in Section 8.3.3 and also the main challenges and opportunities span beyond the current shallow embedding approach and they will be discussed in the next section.

9.3 Further Research Directions

In this section we outline our long-term perspectives, identifying further research paths to expand our understanding of domain-specific modeling, languages and their applications in and beyond the context of self-adaptive software systems engineering and model manipulations.

9.3.1 On Engineering Self-Adaptive Software Systems

At the beginning of Section 9.1 we argued that systems must become adaptive. The main question is therefore, how can we aid software developers and system administrators to create self-adaptive software systems? In the following, we outline some of the possible research directions.

Towards Higher-Level Abstractions The abstraction level provided by FCDL is the one of feedback control loops. Using MDE techniques, new higher-level abstractions can be built on the top of it. For example, Lapouchnian *et al.* [Lapouchnian et al., 2006], Goldsby *et al.* [Goldsby et al., 2008] and Sawyer *et al.* [Sawyer et al., 2007] have used goal models for specifying requirements and behavior of an adaptive systems. Such goal-models could be used as inputs to a model transformation that systematically generates the corresponding FCDL models or xFCDL code. One of the advantages is that through the transformation chain traceability between these models can be maintained so it is possible to follow higher-level goals representation down to the implementation code or even to the runtime. The major challenge is in defining the model refinement from higher-level models into FCDL or xFCDL which has to be further researched.

Towards Model@run.time In the previous section we have mentioned the need for proper error handling and fault tolerance. Model@run.time (*cf.* Section 2.2.1 on page 23) might provide an approachable solution towards this objective. The FCDL instance model could be preserved at runtime with causal links to the underlying adaptive elements. Each adaptive element would then represent a model instance and model operations and properties would correspond to messages. The FCDL invariants including the user-defined ones could therefore be verified at runtime, reusing the very same mechanism that is used at design time. This is also an important feature to enable reliable dynamic reconfiguration of the FCL itself through structural adaptations [Léger et al., 2010].

Towards DSL Composition The xFCDL language allows developers to directly implement the behavior of adaptive elements. However, as we have already noted, Xbase might

be too low level for concepts such as decision tables or rule based expressions. Ideally, developers should be able to choose from a variety of languages and all should have the FCDL concepts available at the right level of abstraction, *i.e.*, the one of the target language. Moreover, the different languages should be available in a modular way, composable together.

The current implementation based on Xtext is limited in language composition. While it supports language mix-ins, in general, it is not possible to guarantee that two language extensions are compatible, because the resulting grammar might be ambiguous. Xtext internally relies on ANTLR[4], which is a two phases LL(*) parser with composition limitations [Bravenboer and Visser, 2008].

On the other hand, composition of domain-specific models and languages is an actively studied area and there exists a number of approaches [Voelter, 2011, Cazzola and Poletti, 2010, Jezequel et al., 2013]. Furthermore, Mussbacher *et al.* [Mussbacher et al., 2012] summarized an overview about composition facilities in different meta-modeling approaches (*e.g.* Kermeta, UML). However, it still remains an open question how to compose different models and languages in a cost-effective and reliable way. We need to further investigate and experiment with different technologies to find what are the feasible approaches to include multiple implementation possibilities in the Actress modeling environment.

9.3.2 On Model Manipulation

Systematic model manipulations, including model transformations and model consistency checking, is one of the essences of MDE as it is used to gradually refine abstract model concepts into concrete implementations. In this thesis we have advocated the internal DSL approach as a viable and lightweight alternative to the external model manipulation DSLs.

Because internal DSLs reuse the infrastructure of the host language, they are significantly easier to develop. However, traditionally it has been difficult to realize domain-specific analysis and optimization [Brown et al., 2011].

In this section we will introduce a perspective direction that should alleviate these issues and further improve the capabilities of internal DSLs.

Lightweight Modular Staging Established by Taha and Sheard [Taha and Sheard, 1997], multi-stage programming is a technique that allows to naturally separate computations into a number of stages that are distinguished by either a frequency of execution or availability of information [Rompf et al., 2013]. A stage acts as a generator that when executed produces a program to be run in the next stage. In Scala, this technique is realized as *Lightweight Modular Staging* (LMS) [Rompf and Odersky, 2010], a library that offers a set of core components for building code generators and embedded compilers in Scala, *i.e.*, techniques for deep DSL embedding. Effectively, it allows one to produce alternative rep-

[4] http://www.antlr.org/

resentations of Scala expressions in a modular fashion (depending which trait is mixed in the class) that are in the later stage reified into new Scala or any other source code.

To better illustrate LMS, in the rest of the section, we give some examples of its possibilities for domain-specific error-checking and optimization in Sigma context.

Towards Automated Checking for Invariant Unsatisfiability One of the problems which becomes apparent when the number of invariants grows is how to make sure that they are not contradictory. For example, consider the following two Sigma invariants:

```
def invLessThan3Ports = self.types forall { x => x.ports.size < 3 }
def invAtLeast3Ports = self.types exists { x => x.ports.size >= 3 }
```

While, it is obvious that they are contradictory, using only the shallow embedding, it is very difficult to automatically realize it at design time. With LMS, on the other hand, invariants can be translated into first-order logic such as $\forall(x)(Port(x) \wedge (ports(x) < 3))$ and $\exists(x)(Port(x) \wedge (ports(x) >= 3))$. The contradiction of the formulae can be then automatically verified by a SMT (Satisfiability Modulo Theories) solver (*e.g.* Yices[5]) [Clavel et al., 2009].

Towards Declarative to Imperative M2M Transformations Another possible LMS application is for M2M transformations. The declarative description of M2M rules is convenient for a developer, but implies certain performance overhead at runtime. Using LMS, we could convert declarative rules into an imperative code. For example, taking the xFCDL to FCDL transformation:

```
def ruleAdaptiveElement(source: AdaptiveElementDecl, target: AdaptiveElementType)
def ruleProperty(source: PropertyDecl, target: Property)
// ..
```

could be converted into the following imperative code with explicit rule scheduling:

```
val it = source.eAllContents
while (it.hasNext) {
  val x = it.next

  if (x.isInstanceOf[AdaptiveElementDecl]) {
    ruleAdaptiveElement(x.asInstanceOf[AdaptiveElementDecl],
      create[AdaptiveElementType])

  else if (x.isInstanceOf[PropertyDecl])
    ruleAdaptiveElement(x.asInstanceOf[PropertyDecl], create[Property])

  // ...

  else
    logger warn (s"No rule to transfrom ${x}.")
}
```

[5] http://yices.csl.sri.com/

Static Text Decorators The SIGMA M2T performance issues identified in Section 8.3.2 could be solved by applying decorators at design time. For example, following excerpt of the M2T transformation between FCDL and Promela (*cf.* Section 7.3.3):

```
!"waiting:" indent {
  !"if"
  for (ic <- elem.contracts) {
    indent { !s"// ${ic}" }
    !"::"
    indent { genActivationCondition(ic) }
  }
  !"fi;"
}
```

could be transformed into:

```
sb append "waiting:\n"
sb append "if\n"
for (ic <- elem.contracts) {
  sb append "\t// ${ic}\n"
  sb append "::\n"
  {
    // inlined genActivationCondition(ic)
    // with all !"<str>" expressions converted to
    // !"\t<str>"
  }
  sb append "\n"
}
sb append "fi;\n"
```

The transformed code has the indent statements unrolled and consecutive template execution inlined. It therefore does not have extra SIGMA-related performance overhead.

It is important to realize, that LMS is far from being a silver bullet to all the issues related to internal DSLs. There are still many challenges to be addressed, for example how to make tools aware of the DSL abstraction (*e.g.*, improving compiler feedback and debugger stack presentation)? How to loosen the syntax restrictions implied by host languages? Moreover, while LMS is being successfully used in practice [Brown et al., 2011], it is still a research work and it is not an easy library to use. However, there is an interesting research possibility with regards to LMS development. Essentially, the optimization and generation from one stage to another involves transformation of *Abstract Syntax Trees* (AST). These are tree models and thus it should be possible to apply some techniques from MDE. Since SIGMA is a Scala internal DSL, it might be a good fit for implementing the AST manipulations.

9.4 Final Remarks

Both, self-adaptive software systems and model-driven engineering are very exciting areas of research. This thesis aims at expanding their state of the art by the contribution summarized earlier in this chapter. At the same time, as we have outlined above, a lot more has to come and there are many more exciting opportunities and challenges in the future research. Nevertheless, one should never forget the main challenge:

> The computing scientist's main challenge is not to get confused by the complexities of his own making.
>
> --- E. W. Dijkstra

Part IV

Appendices

A Publications

- Philippe Collet, Filip Křikava, Johan Montagnat, Mireille Blay-Fornarino, David Manset. Issues and Scenarios for Self-Managing Grid Middleware, *Proceeding of the 2nd workshop on Grids meets autonomic computing (GMAC '10)*, 2010
- Filip Křikava, Philippe Collet, Mireiller Blay-Fornarino.
  Uniform and Model-Driven Engineering of Feedback Control Systems, *Proceeding of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC'11)*, 2011 short paper
- Filip Křikava, Philippe Collet.
  A Reflective Model for Architecting Feedback Control Systems, *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE'11)*, 2011
- Filip Křikava, Philippe Collet.
  On the Use of an Internal DSL for Enriching EMF Models, *Proceedings of the 2012 International Workshop on OCL and Textual Modelling (OCL'12)*, 2012
- Filip Křikava, Philippe Collet, Robert France.
  Actor-based Runtime Model of Adaptable Feedback Control Loops, *Proceedings of the Proceedings of the 7th International Workshop on models@run.time (MRT'12)*, 2012

The Actress modeling environment has been also presented at:

- *EGI Technical Forum* 2012. Filip Křikava, Javier Rojas Balderrama, Johan Montagnat, Philippe Collet. Using Adaptation Strategies to Improve Grid Operations
  Prague, Czech Republic, September 2012
- *Journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR-GPL'12)* 2012. Filip Křikava, Philippe Collet. Using Architecture Models to Rapidly Prototype Feedback Control Systems
  Rennes, France, June 2012

The Sigma model manipulation DSLs have been also presented at:

- *EclipseCon Europe 2012 Modeling Symposium*. Filip Křikava. Enriching EMF Models with Scala
  Ludwigsburg, Germany, October 2012
  `http://www.slideshare.net/krikava/enriching-emf-models-with-scala`
- *4th annual Scala Workshop 2013*. Filip Křikava. Model Manipulation Using Embedded DSLs in Scala
  Montpellier, France, July 2013
  `http://www.slideshare.net/krikava/enriching-emf-models-with-scala`

B FCDL Reference

B.1 FCDL Graphical Notation

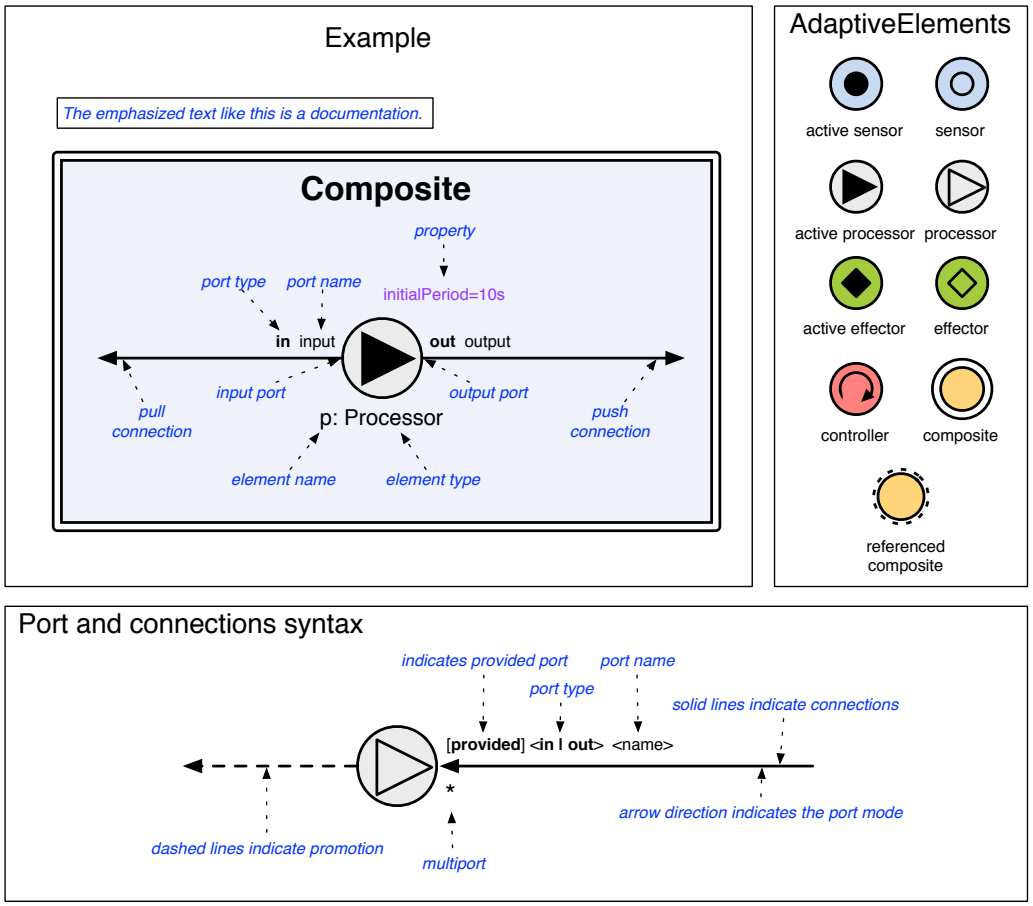


Figure B.1: FCDL Graphical Notation

B.2 FCDL Abstract Syntax: Types Packages

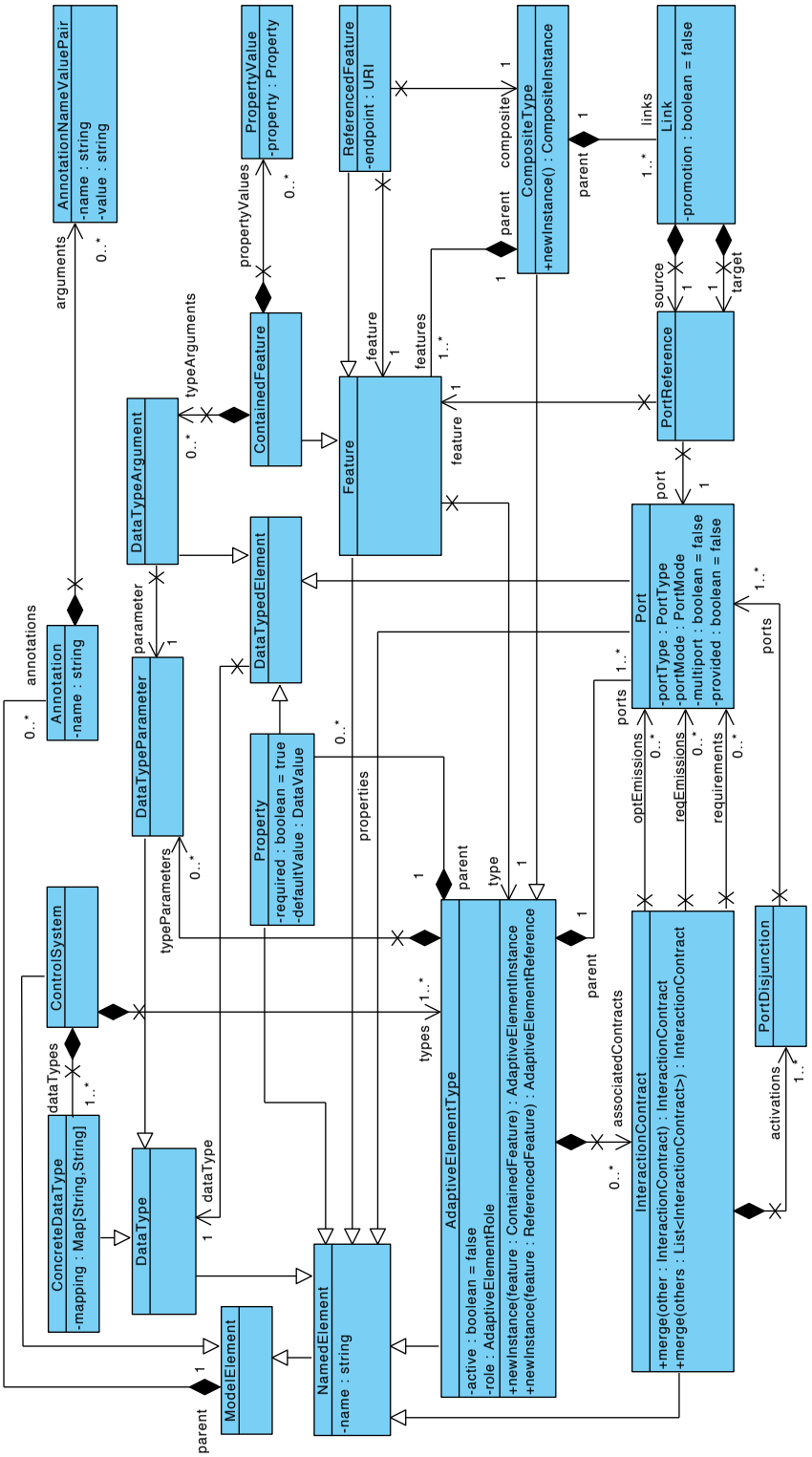


Figure B.2: FCDL Types Package

B.3 FCDL Abstract Syntax: Instances Package

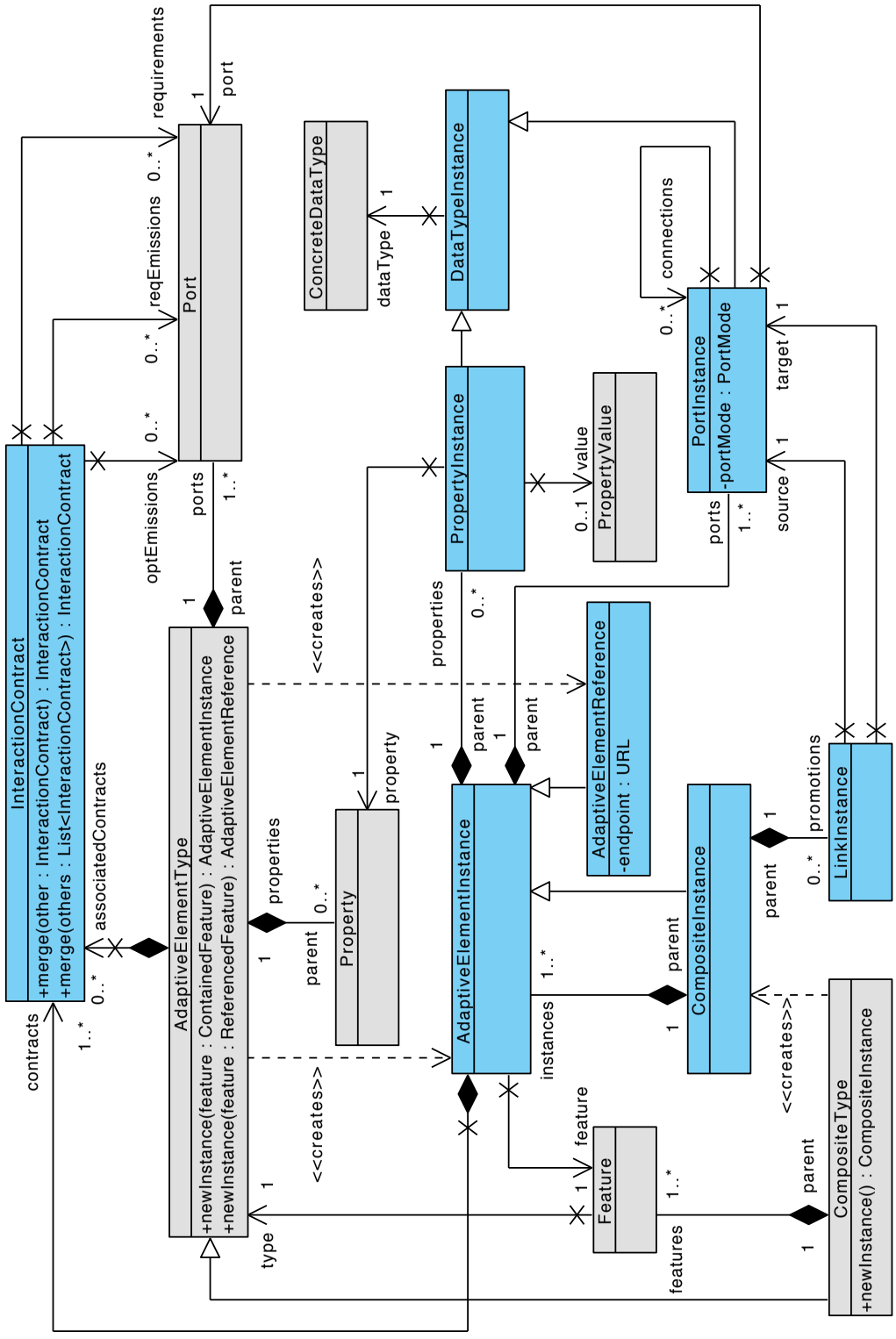


Figure B.3: FCDL Instances Package

B.4 Scala Implementation of Interaction Contract Inference

```
override def getContracts(): EList[InteractionContract] = {

 // returns an execution path start starts in the given port
 def execPath(portInst: PortInstance): Buffer[InteractionContract] = {
  def execPath0(visited: Buffer[PortInstance])(portInst: PortInstance):
    Buffer[InteractionContract] = {

    // mark the current port as visited
    visited += portInst
    // the adaptive element owning this port
    val elem = portInst.parent
    // merged interaction contract associated with this port
    val ic = portInst.contracts.toList match {
      case x :: xs ⇒(x /: xs)(_ merge _)
      case Nil ⇒
        // if the model is well-formed this will never happen
        // (cf contract completeness)
        sys.error(s"No interaction contrcat associated with port $portInst")
    }
    // ports we should visit from this adaptive element
    // all the interaction contract ports
    val toVisit = (ic.requirements ++ ic.emissions ++
      ic.activation.map(_.ports).flatten)
    // interaction contracts are defined at the type level
    // we have to therefore map the Port into PortInstance
    val toVisitInst = toVisit map { p ⇒elem.getPort(p) }
    // skip the already visited ports
    val toVisitNew = toVisitInst -- visited

    // recursively continue the graph traversal
    val next = toVisitNew map { p ⇒
      val paths = p.connections map execPath0(visited)
      paths.flatten
    }
    ic +: next.flatten
  }

  // start the recursive graph traversal at the given port
  execPath0(Buffer())(portInst)
 }

 def promote(ic: InteractionContract) = {
  def promotion: PartialFunction[Port, Port] = {
    case p if this.promotions.exists(_.source.port == p) ⇒
      this.promotions.find(_.source.port == p).map(_.target.port).get
  }

  val A = ic.activation.map { disj ⇒
    val ports = disj.ports.collect(promotion)
    PortDisjunction(ports: _*)
  }.filter(_.ports.nonEmpty)

  val R = ic.requirements collect promotion
  val Ereq = ic.reqEmissions collect promotion
  val Eopt = ic.optEmissions collect promotion

  InteractionContract(activation = A, requirements = R,
    reqEmissions = Ereq, optEmissions = Eopt)
 }

 val startNodes = this.promotions map (_.source) collect {
  case p @ PortInstance(INPUT, PUSH) ⇒p
  case p @ PortInstance(OUTPUT, PULL) ⇒p
  case p @ PortInstance(SELF, PUSH) ⇒p
 }

 // Step 1: find execution paths
 val execPaths = startNodes map execPath
```

```
69    // Step 2: merge
70    val merged = execPaths map {
71      _ match {
72        case Buffer(x, xs @ _*) ⇒(x /: xs)(_ merge _)
73      }
74    }
75    // Step 3: promote distinct IC
76    val promoted = merged.distinct map promote
77    // Step 4: remove empty ones
78    val paths = promoted.collect {
79      case ic if ic.activation.exists(_.ports.nonEmpty) ⇒ic
80    }
81    paths
82  }
```

Listing B.1: Scala implementation of the interaction contract inference

C xFCDL Reference

C.1 Abstract Syntax

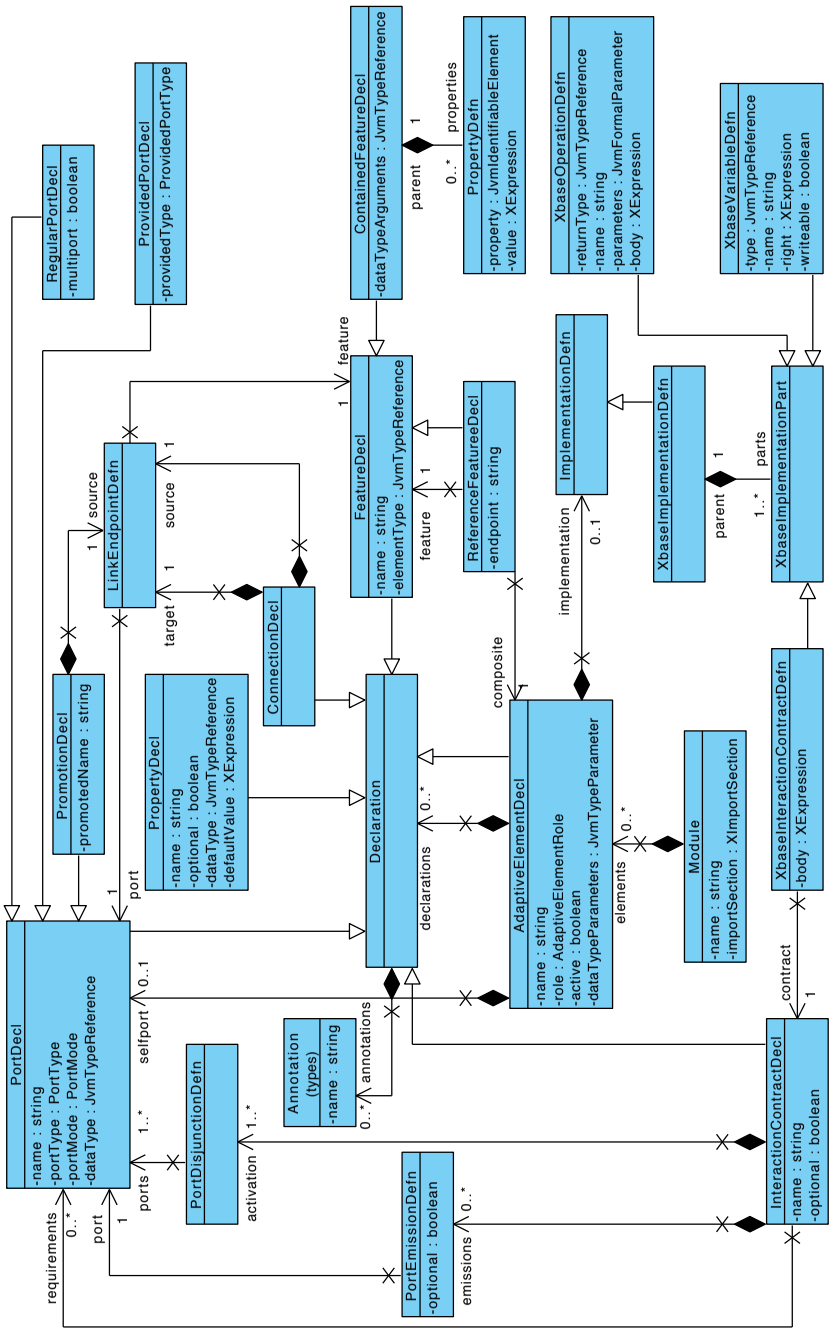


Figure C.1: xFCDL Abstract Syntax

C.2 Concrete Syntax

Following is the listing of the xFCDL concrete syntax using the Xtext grammar language[1] which is a simple EBNF-like DSL for defining language grammars.

```
grammar fr.unice.i3s.actress.modeling.xfcdl.XFCDL with org.eclipse.xtext.xbase.Xbase

import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
import "http://actress.i3s.unice.fr/modeling/xfcdl/0.1"
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://actress.i3s.unice.fr/modeling/fcdl/1.0/fcdl/types" as fcdl

// -----------------------------------------------------------------------------
// STRUCTURE
// -----------------------------------------------------------------------------
Module:
   ('package' name=QualifiedName)?
   importSection=XImportSection?
   (elements+=AdaptiveElementDecl)*;

AdaptiveElementDecl:
   (annotations+=Annotation)*
   (active?='active')? role=AdaptiveElementRole name=ValidID
   ('<' dataTypeParameters+=JvmTypeParameter (',' dataTypeParameters+=JvmTypeParameter)* '>')?
   '{'
   (declarations+=Declaration)*
   (implementation=ImplementationDefn)?
   '}';

Annotation returns fcdl::Annotation:
   '@' name=ValidID (=>'(' arguments+=AnnotationArgument (',' arguments+=AnnotationArgument)*
       ')')?
;

AnnotationArgument returns fcdl::AnnotationNameValuePair:
   name=ValidID '=' value=STRING
;

Declaration:
   FeatureDecl
   | PropertyDecl
   | PortDecl
   | PromotionDecl
   | ConnectionDecl
   | InteractionContractDecl;

FeatureDecl:
    (ContainedFeatureDecl | ReferencedFeatureDecl);

ContainedFeatureDecl:
   (annotations+=Annotation)*
   'feature' name=ValidID '=' 'new' elementType=JvmTypeReference ('{' (properties+=
       PropertyDefn)* '}')?;

ReferencedFeatureDecl:
   (annotations+=Annotation)*
   'feature' name=ValidID '=' 'ref' composite=[AdaptiveElementDecl] '.' feature=[FeatureDecl]
       '@' endpoint=STRING;

PropertyDefn:
   property=[types::JvmIdentifiableElement|ValidID] '=' value=XExpression;

PropertyDecl:
   (annotations+=Annotation)*
   (optional?='optional')? 'property' name=ValidID ':' dataType=JvmTypeReference ('='
       defaultValue=XExpression)?;

PortDecl:
```

[1] http://www.eclipse.org/Xtext/documentation.html#grammarLanguage

```
60    RegularPortDecl
61    | ProvidedPortDecl;
62
63  RegularPortDecl:
64    (annotations+=Annotation)*
65    ((portMode=PortMode)? & (portType=PortType)?)
66    (multiport?='multiport' | 'port')
67    name=ValidID ':'
68    dataType=JvmTypeReference;
69
70  ProvidedPortDecl:
71    (annotations+=Annotation)*
72    'provided' providedType=ProvidedPortType name=ValidID ':' dataType=JvmTypeReference;
73
74  ConnectionDecl:
75    (annotations+=Annotation)*
76    'connect' source=LinkEndpointDefn 'to' target=LinkEndpointDefn;
77
78  PromotionDecl:
79    (annotations+=Annotation)*
80    'promote' source=LinkEndpointDefn ('as' promotedName=ValidID)?;
81
82  InteractionContractDecl:
83    (annotations+=Annotation)*
84    'act' name=ValidID '('
85    activation+=PortDisjunctionDefn (',' activation+=PortDisjunctionDefn)* ';'
86    (requirements+=[PortDecl] (',' requirements+=[PortDecl])*)? ';'
87    (emissions+=PortEmissionDefn (',' emissions+=PortEmissionDefn)*)? ')' (optional?='?')?;
88
89  LinkEndpointDefn:
90    feature=[FeatureDecl|ValidID] ('.') port=[PortDecl|ValidID];
91
92  PortDisjunctionDefn:
93    ports+=[PortDecl] ('v' ports+=[PortDecl])*;
94
95  PortEmissionDefn:
96    port=[PortDecl] (optional?='?')?;
97
98  ImplementationDefn:
99    'implementation' XbaseImplementationDefn;
100
101 // ------------------------------------------------------------------------
102 // XBASE Implementation
103 // ------------------------------------------------------------------------
104 XbaseImplementationDefn:
105   {XbaseImplementationDefn} 'xbase' '{'
106   (parts+=XbaseImplementationPart)*
107   '}';
108
109 XbaseImplementationPart:
110   XbaseVariableDefn
111   | XbaseInteractionContractDefn
112   | XbaseOperationDefn;
113
114 XbaseInteractionContractDefn:
115   'act' contract=[InteractionContractDecl|ValidID] body=XBlockExpression;
116
117 XbaseOperationDefn:
118   'def' name=ValidID ('(' parameters+=FullJvmFormalParameter (',' parameters+=
          FullJvmFormalParameter)* ')')?
119   (':' returnType=JvmTypeReference)? body=XBlockExpression;
120
121 XbaseVariableDefn:
122   (writeable?='var' | 'val') (=> (name=ValidID ':' type=JvmTypeReference) | name=ValidID)
123   ('=' right=XExpression)?;
124
125 XVariableDeclaration returns xbase::XExpression:
126   {xbase::XVariableDeclaration} (writeable?='var' | 'val') (=> (name=ValidID ':' type=
          JvmTypeReference) | name=ValidID)
127   ('=' right=XExpression)?;
128
```

```
129 JvmFormalParameter returns types::JvmFormalParameter:
130    name=ValidID => (':' parameterType=JvmTypeReference)?;
131
132 FullJvmFormalParameter returns types::JvmFormalParameter:
133    name=ValidID ':' parameterType=JvmTypeReference;
134
135 // -------------------------------------------------------------------------
136 // ENUMS
137 // -------------------------------------------------------------------------
138 enum AdaptiveElementRole:
139    SENSOR='sensor'
140    | PROCESSOR='processor'
141    | CONTROLLER='controller'
142    | EFFECTOR='effector'
143    | COMPOSITE='composite';
144
145 enum PortMode returns fcdl::PortMode:
146    PULL='pull'
147    | PUSH='push';
148
149 enum PortType returns fcdl::PortType:
150    INPUT='in'
151    | OUTPUT='out';
152
153 enum ProvidedPortType:
154    SENSOR='sensor'
155    | EFFECTOR='effector';
```

Listing C.1: xFCDL Concrete Syntax

C.3 xFCDL to FCDL Transformation Rules

There are two concepts in the xFCDL language that have no correspondence in FCDL: Xbase implementation block and Xbase expression that are used for specifying property values. Both concepts are only used in connection with a code generation and therefore similarly to what is done with data type conversion, we only store references to the source elements using the EMF adapters to be used by the code generator.

Following is the list of the transformation rules with some additional details in the cases where the mapping is not straight-forward.

- `Module` maps to `ControlSystem`
- `AdaptiveElement` whose role is set to *composite* maps to `CompositeType`, others map to `AdaptiveElementType`.
- `JvmTypeParameter` maps into `DataTypeParameter` as discussed in Section 5.1.5.
- `JvmTypeReference` maps into `DataType` as discussed in Section 5.1.5. `JvmTypeReference` used for data type arguments in contained feature definitions maps into `DataTypeArgument`.
- `ContainedFeatureDecl` maps into `ContainedFeature`.
- `ReferencedFeatureDecl` maps into `ReferencedFeature`.
- `PropertyDecl` maps into `Property`. In xFCDL the property default value is encoded as a Xbase expression which cannot be evaluated at the transformation time and therefore a default value is therefore not used. The expression is only attached to the `Property` so it can be later used by a code generator.
- `RegularPortDecl` maps into `Port`.
- `ProvidedPortDecl` maps into `Port` with *provided* property set to **true**.

- `LinkEndpointDefn` maps into `PortReference`.
- `ConnectionDecl` maps into `Link`.
- `PromotionDecl` maps into `Link` with *promotion* property set to **true**.
- `PropertyDefn` maps into `PropertyValue` however just like in the case of `PropertyDecl` the actual value is stored as Xbase expression. It is therefore not used directly, but only attached to be used later by a code generator.
- `PortDisjunctionDefn` maps into `PortDisjunction`.
- `InteractionContractDecl` maps into `InteractionContract`. The *reqEmissions* and *optEmissions* are populated based on the *optional* property of the `PortEmissionDefn` referenced from the `InteractionContractDecl`.

D Running Scenario Implementation

D.1 Running Example xFCDL Definitions

```
package demo.webserver

active sensor FileTailer {
   out push port lines: String
   port selfport: String

   property file: String

   act activate(selfport;;lines)
}

processor AccessLogParser {
   in push port lines: String
   out push port size: long
   out push port requests: int

   act activate(lines;;requests,size)
}

effector ContentAdaptor {
   in push port contentTree: double

   act activate(contentTree;;)
}

controller UtilizationController {
   in push port utilization: double
   out push port contentTree: double

   property k: double
   property targetUtilization: double // U^*
   property M: int

   act activate(utilization;;contentTree)

   implementation xbase {
      var G = M

      act activate {
         val E = targetUtilization - utilization

         G = G + k * E
         if (G < 0) G = 0
         if (G > M) G = M

         G
      }
   }
}

processor Accumulator {
   in push port input: long
   out pull port sum: long

   act onInput(input;;)
   act onSum(sum;;)

   implementation xbase {
      var value: Long = 0L

```

```
61        act onInput {
62          value = value + input
63        }
64
65        act onSum {
66          value
67        }
68     }
69  }
70
71  processor LoadMonitor {
72     in pull port requests: int
73     in pull port size: long
74     out pull port utilization: double
75
76     property a: Double
77     property b: Double
78
79     act activate(utilization;requests,size;)
80
81     implementation xbase {
82        var lastTime = System::currentTimeMillis
83
84        act activate {
85           val elapsed = System::currentTimeMillis - lastTime
86           val R = requests.get.get / elapsed
87           val W = size.get.get / elapsed
88
89           a*R + b*W
90        }
91     }
92  }
93
94  active processor PeriodicTrigger<T> {
95     push out port output: T
96     pull in port input: T
97     port selfport: long
98
99     provided sensor period: Duration
100    provided effector setPeriod: Duration
101
102    property initialPeriod: Duration
103
104    act activate(selfport; input; output?)
105    act onSetPeriod(setPeriod;;period?)
106
107    implementation xbase {
108
109       var currentPeriod = initialPeriod
110       var task: Cancellable
111
112       def init {
113          reschedule
114       }
115
116       def destroy {
117          task.cancel
118       }
119
120       act activate {
121         log.info("Activate at "+selfport.get)
122
123         switch data : input.get {
124           case data.present : output.put(data.get)
125           default : log.info("No data available on the input port")
126         }
127       }
128
129       act onSetPeriod {
130          if (setPeriod != currentPeriod) {
131            currentPeriod = setPeriod
```

```
132             reschedule()
133             period.put(currentPeriod)
134          }
135       }
136
137       def reschedule {
138          task = context.scheduler.schedule(2.seconds, currentPeriod) [|
139             selfport.put(System::currentTimeMillis)
140          ]
141       }
142    }
143 }
144
145 composite ApacheWebServer {
146    property accessLogFile: String
147
148    feature accessLog = new FileTailer {
149     file = accessLogFile
150    }
151    feature accessLogParser = new AccessLogParser
152    feature adaptor = new ContentAdaptor
153
154    connect accessLog.lines to accessLogParser.lines
155
156    promote accessLogParser.size
157    promote accessLogParser.requests
158    promote adaptor.contentTree
159 }
160
161 composite UtilizationMonitor {
162    property a: double
163    property b: double
164
165    feature requestsCounter = new Accumulator
166    feature responseSizeCounter = new Accumulator
167    feature loadMonitor = new LoadMonitor {
168     a = this.a
169     b = this.b
170    }
171
172    connect requestsCounter.sum to loadMonitor.requests
173    connect responseSizeCounter.sum to loadMonitor.size
174
175
176    promote requestsCounter.input as requests
177    promote responseSizeCounter.input as size
178    promote loadMonitor.utilization
179 }
180
181 composite QOSControl {
182
183    property k: double
184    property targetUtilization: double
185    property M: int
186    property a: double
187    property b: double
188
189    feature utilization = new UtilizationMonitor {
190       a = this.a
191       b = this.b
192    }
193
194    feature scheduler = new PeriodicTrigger<Double> {
195       initialPeriod = 10.seconds
196    }
197
198    feature utilController = new UtilizationController {
199       k = this.k
200       targetUtilization = this.targetUtilization
201       M = this.M
202    }
```

```
203
204     connect utilization.utilization to scheduler.input
205     connect scheduler.output to utilController.utilization
206
207     promote utilization.requests
208     promote utilization.size
209     promote utilController.contentTree
210 }
211
212 @Main
213 composite ApacheQOS {
214     property k: double
215     property targetUtilization: double
216     property M: int
217     property a: double
218     property b: double
219     property accessLogFile: String
220
221     feature apache = new ApacheWebServer {
222       accessLogFile = this.accessLogFile
223     }
224
225     feature control = new QOSControl {
226       k = this.k
227       targetUtilization = this.targetUtilization
228       M = this.M
229       a = this.a
230       b = this.b
231     }
232
233     connect apache.requests to control.requests
234     connect apache.size to control.size
235     connect control.contentTree to apache.contentTree
236 }
```

Listing D.1: xFCDL code of the running example as shown in Figure 3.8

D.2 `PeriodicTrigger` Implementation

D.2.1 Adaptive Element Delegate

```
1  package demo.webserver;
2
3  import fr.unice.i3s.actress.modeling.fcdl.japi.AdaptiveElementDelegate;
4  import fr.unice.i3s.actress.modeling.fcdl.japi.DefaultInPullPort;
5  import fr.unice.i3s.actress.modeling.fcdl.japi.DefaultInPushPort;
6  import fr.unice.i3s.actress.modeling.fcdl.japi.DefaultOutPushPort;
7  import fr.unice.i3s.actress.modeling.fcdl.japi.DefaultSelfPort;
8  import fr.unice.i3s.actress.modeling.fcdl.japi.DirectorContext;
9  import fr.unice.i3s.actress.modeling.fcdl.japi.Duration;
10 import fr.unice.i3s.actress.modeling.fcdl.japi.InPullPort;
11 import fr.unice.i3s.actress.modeling.fcdl.japi.InPushPort;
12 import fr.unice.i3s.actress.modeling.fcdl.japi.OutPushPort;
13 import fr.unice.i3s.actress.modeling.fcdl.japi.PullWrapper;
14 import fr.unice.i3s.actress.modeling.fcdl.japi.PushWrapper;
15 import fr.unice.i3s.actress.modeling.fcdl.japi.SelfPort;
16
17 @SuppressWarnings("all")
18 public class PeriodicTrigger<T> extends AdaptiveElementDelegate {
19   protected final PeriodicTriggerAct<T> _act;
20
21   protected final Duration initialPeriod;
22
23   protected final OutPushPort<T> output;
24
25   protected final InPullPort<T> input;
26
27   protected final SelfPort<Long> selfport;
```

```
28
29  protected final OutPushPort<Duration> period;
30
31  protected final InPushPort<Duration> setPeriod;
32
33  public PeriodicTrigger(final DirectorContext context,
34     final Duration initialPeriod) {
35    super(context);
36
37    this.initialPeriod = initialPeriod;
38    this._act = new PeriodicTriggerAct<T>(context, initialPeriod);
39    this.output = new DefaultOutPushPort(context);
40    this.input = new DefaultInPullPort(context);
41    this.selfport = new DefaultSelfPort(context);
42    this.period = new DefaultOutPushPort(context);
43    this.setPeriod = new DefaultInPushPort(context);
44  }
45
46  @Override
47  public void init() {
48    _act.init();
49  }
50
51  @Override
52  public void destroy() {
53    _act.destroy();
54  }
55
56  @Override
57  public boolean preActivate() {
58    if (setPeriod.nonEmpty()) {
59      // act onSetPeriod(setPeriod; ; period?)
60      return true;
61    }
62    if (!selfport.atLeastOneNonEmpty()) {
63      // act activate(selfport; input; output?)
64      return true;
65    }
66    return false;
67  }
68
69  @Override
70  public void activate() {
71    if (setPeriod.isEmpty()) {
72      // act onSetPeriod(setPeriod; ; period?)
73      Duration setPeriodValue = setPeriod.get();
74      PushWrapper<Duration> periodPush = new PushWrapper<Duration>(period);
75      _act.setPeriod(setPeriodValue, periodPush);
76    } else if (selfport.isEmpty()) {
77      // act activate(selfport; input; output?)
78      Long selfportValue = selfport.get();
79      PullWrapper<T> inputPull = new PullWrapper<T>(input);
80      PushWrapper<T> outputPush = new PushWrapper<T>(output);
81      _act.activate(selfportValue, inputPull, outputPush);
82    } else {
83      throw new IllegalStateException("Invalid execution");
84    }
85  }
86
87  @Override
88  public String toString() {
89    return "PeriodicTrigger";
90  }
91 }
```

Listing D.2: Synthesized `PeriodicTrigger` adaptive element delegate

D.2.2 Adaptive Element Act

```
@SuppressWarnings("all")
public class PeriodicTriggerAct<T> extends AdaptiveElementAct {
  private final Duration initialPeriod;

  private Duration currentPeriod = this.initialPeriod;

  private Cancellable task;

  private final Push<Long> selfport;

  public PeriodicTriggerAct(final DirectorContext context,
     final Duration initialPeriod) {
    super(context);
    this.initialPeriod = initialPeriod;
    this.selfport = new PushSelfPort(context);
    this.currentPeriod = this.initialPeriod;

  }

  @Override
  public void init() {
    this.reschedule();
  }

  @Override
  public void destroy() {
    this.task.cancel();
  }

  public Cancellable reschedule() {
    DirectorContext _context = this.getContext();
    Scheduler _scheduler = _context.getScheduler();
    Duration _seconds = Duration.seconds(2);
    final Runnable _function = new Runnable() {
      public void run() {
        long _currentTimeMillis = System.currentTimeMillis();
        PeriodicTriggerActs.this.selfport.put(Long
            .valueOf(_currentTimeMillis));
      }
    };
    Cancellable _schedule = _scheduler.schedule(_seconds,
       this.currentPeriod, _function);
    Cancellable _task = this.task = _schedule;
    return _task;
  }

  protected void activate(final long selfport, final Pull<T> input,
     final Push<T> output) {
    String _plus = ("Activate at " + Long.valueOf(selfport));
    this.log.info(_plus);
    Optional<T> _get = input.get();
    final Optional<T> data = _get;
    boolean _matched = false;
    if (!_matched) {
      boolean _isPresent = data.isPresent();
      if (_isPresent) {
        _matched = true;
        T _get_1 = data.get();
        output.put(_get_1);
      }
    }
  }

  protected void onSetPeriod(final Duration setPeriod,
     final Push<Duration> period) {
    boolean _notEquals = (!Objects.equal(setPeriod, this.currentPeriod));
    if (_notEquals) {
      this.currentPeriod = setPeriod;
      this.reschedule();
      period.put(this.currentPeriod);
```

```
    }
  }
}
```

Listing D.3: Synthesized `PeriodicTrigger` adaptive element act

D.3 `ApacheQOS` Composite Launcher

```
public class ApacheQOSLauncher {

  public static ApacheQOS createCompositeDelegate() {
    double k = 0D;
    double targetUtilization = 0D;
    int M = 0;
    double a = 0D;
    double b = 0D;

    return new ApacheQOS(k, targetUtilization, M, a, b);
  }

  public static void main(String[] args) {
    ActressSystem system = new ActressSystem("ApacheQOS");
    system.execute(createCompositeDelegate());
  }

}
```

Listing D.4: Synthesized`ApacheQOS` launcher

D.4 `UtilizationMonitor` Composite Delegate

```
public class UtilizationMonitor extends CompositeActorDelegate {

  protected final double a;
  protected final double b;

  public AdaptiveElementRef requestsCounter;
  public AdaptiveElementRef sizeCounter;
  public AdaptiveElementRef loadMonitor;

  public UtilizationMonitor(final double a, final double b) {
    this.a = a;
    this.b = b;
  }

  private Accumulator createRequestsCounter(final DirectorContext context) {
    return new Accumulator(context);
  }

  private Accumulator createSizeCounter(final DirectorContext context) {
    return new Accumulator(context);
  }

  private LoadMonitor createLoadMonitorDelegate(final DirectorContext context) {
    final java.lang.Double _a = this.a;
    final java.lang.Double _b = this.b;
    return new LoadMonitor(context, _a, _b);
  }

  @Override
  protected void startContainedFeatures() {
    requestsCounter = startAdaptiveElement("requestsCounter",
       new DelegateCreator() {
        @Override
        public AdaptiveElementDelegate create(
```

```
35            final DirectorContext director) {
36          return createRequestsCounter(director);
37        }
38       });
39    sizeCounter = startAdaptiveElement("sizeCounter",
40       new DelegateCreator() {
41        @Override
42        public AdaptiveElementDelegate create(
43           final DirectorContext director) {
44          return createSizeCounter(director);
45        }
46       });
47    loadMonitor = startAdaptiveElement("loadMonitor",
48       new DelegateCreator() {
49        @Override
50        public AdaptiveElementDelegate create(
51           final DirectorContext director) {
52          return createLoadMonitorDelegate(director);
53        }
54       });
55  }
56
57  @Override
58  protected void lookupReferenceFeatures() {
59    // this composite does not contain any feature references
60  }
61
62  @Override
63  public void connectPorts() {
64    connectPort(loadMonitor, "requests", requestsCounter, "sum");
65    connectPort(loadMonitor, "size", sizeCounter, "sum");
66  }
67
68  @Override
69  protected void promotePorts() {
70    promotePort(requestsCounter, "request", "requests");
71    promotePort(sizeCounter, "request", "requests");
72    promotePort(loadMonitor, "utilization", "utilization");
73  }
74
75  @Override
76  protected void initializeFeatures() {
77    initalizeFeature(requestsCounter);
78    initalizeFeature(sizeCounter);
79    initalizeFeature(loadMonitor);
80  }
81
82  @Override
83  public String toString() {
84    return "UtilizationMonitor";
85  }
86
87 }
```

Listing D.5: Synthesized`UtilizationMonitor` composite delegate

E Experimental Case Studies Implementation

Following is the corresponding xFCDL code implementing the experimental case studies introduce in Section 8.1. For the brevity, we omit the activation methods implementations of the HTCondor touchpoints.

E.1 Case Study 1: HTCondor Local Job Submission Overload Control

E.1.1 xFCDL Code

```
package demo.cs1

import java.util.LinkedList
import demo.common.ProcInfo
import demo.common.CondorInfo
import demo.webserver.PeriodicTrigger

sensor CondorQueueStats {
  out pull port output: int

  property condorConfig: String
  property queueType: String

  provided sensor execTime: long

  act activate(output;;execTime)
}

sensor CondorServiceRate {
  out pull port output: double

  property condorConfig: String

  act activate(output;;)
}

composite Schedd {
  property condorConfig: String

  feature queueSize = new CondorQueueStats {
    condorConfig = this.condorConfig
    queueType = "idle"
  }

  feature serviceRate = new CondorServiceRate {
    condorConfig = this.condorConfig
  }

  promote queueSize.output as queueSize
  promote serviceRate.output as serviceRate
  promote queueSize.execTime
}

processor MovingAverage {
  property initialWindowSize : int = 5

  in port input: double
  out port output: double

  act activate(input;;output)

  implementation xbase {
```

```
53    val values = new LinkedList<Double>()
54     var sum = 0D
55
56    act activate {
57     values.offer(input)
58        sum = sum + input
59        if (values.size == initialWindowSize) {
60         sum = sum - values.poll
61        }
62        sum / values.size
63    }
64   }
65 }
66
67 sensor ProcessCounter {
68   out pull port output: int
69
70   property procFilter: (ProcInfo)=>boolean
71
72   act activate(output;;)
73 }
74
75 processor Aggregate {
76   out pull port output: CondorInfo
77   in pull port queueSize: int
78   in pull port serviceRate: double
79   in pull port dagmanCount: int
80
81   act activate(output; queueSize, serviceRate, dagmanCount;)
82
83   implementation xbase {
84    act activate {
85     new CondorInfo(queueSize.get.get, serviceRate.get.get, dagmanCount.get.get)
86    }
87   }
88 }
89
90 composite CondorStats {
91   property condorConfig: String
92
93   feature schedd = new Schedd {
94    condorConfig = this.condorConfig
95   }
96
97   feature queueSizeAvg = new MovingAverage
98   feature serviceRateAvg = new MovingAverage
99   feature dagmanCounter = new ProcessCounter {
100   procFilter = [|name == "condor_dagman"]
101  }
102  feature aggregate = new Aggregate
103
104  connect schedd.queueSize to queueSizeAvg.input
105  connect schedd.serviceRate to serviceRateAvg.input
106  connect queueSizeAvg.output to aggregate.queueSize
107  connect serviceRateAvg.output to aggregate.serviceRate
108  connect dagmanCounter.output to aggregate.dagmanCount
109
110  promote schedd.execTime
111  promote aggregate.output as condorStats
112 }
113
114 processor PeriodController {
115   push in port input: double
116   push out port output: long
117
118   act activate(input;;output)
119 }
120
121 composite AdaptiveMonitor {
122   feature inputAvg = new MovingAverage
123   feature periodController = new PeriodController
```

```
124
125   connect inputAvg.output to periodController.input
126   promote inputAvg.input
127   promote periodController.output
128 }
129
130 effector FileWriter {
131   pull in port input: String
132
133   property file: String
134
135   act activate(input;;)
136 }
137
138 composite AdaptiveCondorStats {
139   property condorConfig: String
140
141   feature condorStats = new CondorStats {
142     condorConfig = this.condorConfig
143   }
144   feature monitor = new AdaptiveMonitor
145   feature trigger = new PeriodicTrigger<CondorInfo>
146
147   connect condorStats.condorStats to trigger.input
148   connect condorStats.execTime to monitor.input
149   connect monitor.output to trigger.setPeriod
150
151   promote trigger.output as condorStats
152 }
153
154 controller LoadController {
155   push in port input: CondorInfo
156   push out port output: long
157
158   property Nstar: double
159   property Nc: double
160   property p: int
161   property rho0: int
162
163   act activate(input;;output)
164
165   implementation xbase {
166
167     act activate {
168       // getting closer to the notation
169     val N = input.queueSize
170     val sr = input.serviceRate
171     val m = input.dagmanCount;
172     val t = System::currentTimeMillis()
173
174     if (N == 0) {
175     // no op if queue is empty
176     0
177     } else {
178     val rho = switch N {
179       case N < Nstar: rho0 + N * (1 - rho0) / Nstar
180       case N > Nstar: Math::pow(N - Nc, p) / Math::pow(Nstar - Nc, p)
181       default: 1
182     }
183
184     val d = m / (rho * sr)
185     Math::round(d);
186     }
187     }
188   }
189 }
190
191 @Main
192 composite LocalControl {
193   property condorConfig: String
194   property dagmanDelayFile: String
```

```
  property Nstar: double
  property Nc: double
  property p: int
  property rho0: int

  feature condorStats = new AdaptiveCondorStats {
    condorConfig = this.condorConfig
  }
  feature loadController = new LoadController {
     Nstar = this.Nstar
     Nc = this.Nc
     p = this.p
     rho0 = this.rho0
  }
  feature dagmanDelayer = new FileWriter {
    file = dagmanDelayFile
  }

  connect condorStats.condorStats to loadController.input
  connect loadController.output to dagmanDelayer.input

}
```

Listing E.1: xFCDL code of the experimental case study from Section 8.1.2

E.1.2 Interaction Contracts

Adaptive Element Type	Interaction Contract
`AdaptiveCondorStats`	$\langle\, self; \emptyset; \Uparrow (\texttt{condorStats}, \texttt{execTime}) \,\rangle$
`AdaptiveMonitor`	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle$
`Aggregate`	$\langle \Downarrow (\texttt{output}); \Downarrow (\texttt{queueSize}, \texttt{serviceRate}, \texttt{dagmanCount}); \emptyset \rangle$
`CondorQueueStats`	$\langle \Downarrow (\texttt{output}); \emptyset; \Uparrow (\texttt{execTime}) \rangle$
`CondorServiceRate`	$\langle \Downarrow (\texttt{output}); \emptyset; \emptyset \rangle$
`CondorStats`	$\langle \Downarrow (\texttt{condorStats}); \emptyset; \Uparrow (\texttt{execTime}) \rangle$
`FileWriter`	$\langle \Uparrow (\texttt{output}); \emptyset; \emptyset \rangle$
`LoadController`	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle$
`MovingAverage`	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle$
`PeriodController`	$\langle \Uparrow (\texttt{input}); \emptyset; \Uparrow (\texttt{output}) \rangle$
`ProcessCounter`	$\langle \Downarrow (\texttt{output}); \emptyset; \emptyset \rangle$
`Schedd`	$\langle \Downarrow (\texttt{queueSize}); \emptyset; \Uparrow (\texttt{execTime}) \rangle$

Figure E.1: Interaction contracts

E.2 Case Study 2: HTCondor Distributed Job Submission Overload Control

E.2.1 xFCDL Code

```
package demo.cs2

import demo.common.CondorInfo
import demo.cs1.AdaptiveCondorStats
import demo.cs1.FileWriter
```

```
import java.util.LinkedList

controller LoadController {
  push in port localStats: CondorInfo
  pull in port remoteStats: CondorInfo
  push out port output: long

  property Nstar: double
  property Nc: double
  property p: int
  property rho0: int
  property k: int

  act activate(localStats;remoteStats;output)

  implementation xbase {

     var N1_c = 0
     var N1_l = 0

     act activate {
     // getting closer to the notation
     val N_c = remoteStats.get.get.queueSize
     val mu_c = remoteStats.get.get.serviceRate
     val N_l = localStats.queueSize
     val mu_l = localStats.serviceRate

     val r_c = utilizationRatio(N_c,N1_c,mu_c)
     val r_l = utilizationRatio(N_l,N1_l,mu_l)

     N1_c = N_c
     N1_l = N_l

     k * (Math::ceil(Math::min(r_l, r_c)) as int)
   }

     def utilizationRatio(N: int, N1: int, mu: double): double {
        val dN = N - N1
        val dlambda = if (dN > 0) dN + mu else 0

        val rho = dlambda / mu
        val rho_star = switch N {
          case N == 0: rho0
          case N < Nstar: rho0 + N * (1 - rho0) / Nstar
          case N > Nstar: Math::pow(N - Nc, p) / Math::pow(Nstar - Nc, p)
        }

        rho_star / rho
      }
   }
}

processor Queue<T> {
  push in port input: T
  pull out port output: T

  property size: int

  act onPush(input;;)
  act onPull(output;;)

  implementation xbase {

    val queue = new LinkedList<T>()

    act onPush {
        if (queue.size == size) {
          queue.poll
        } else {
          queue.offer(input)
        }
```

```
77    }
78
79    act onPull {
80      queue.poll
81    }
82   }
83 }
84
85 @Main
86 composite CentralStats {
87   property condorConfig: String
88
89   feature condorStats = new AdaptiveCondorStats {
90     condorConfig = this.condorConfig
91   }
92   feature stats = new Queue<CondorInfo> {
93     size = 1
94   }
95
96   connect condorStats.condorStats to queue.input
97 }
98
99 @Main
100 composite LocalControl {
101   property condorConfig: String
102   property dagmanDelayFile: String
103   property Nstar: double
104   property Nc: double
105   property p: int
106   property rho0: int
107   property k: int
108
109   feature condorStats = new AdaptiveCondorStats {
110     condorConfig = this.condorConfig
111   }
112   feature loadController = new LoadController {
113      Nstar = this.Nstar
114      Nc = this.Nc
115      p = this.p
116      rho0 = this.rho0
117      k = this.k
118   }
119   feature dagmanDelayer = new FileWriter {
120     file = dagmanDelayFile
121   }
122   feature centralStats = ref CentralStats.stats @ " akka.tcp://actress@paramount-22.rennes.
          grid5000.fr/user/CentralStats/stats"
123
124   connect condorStats.condorStats to loadController.localStats
125   connect centralStats.output to loadController.remoteStats
126   connect loadController.output to dagmanDelayer.input
127 }
```

Listing E.2: xFCDL code of the experimental case study from Section 8.1.3

E.2.2 Interaction Contracts

Adaptive Element Type	Interaction Contract
`Queue`	$\langle \Downarrow (\texttt{output}); \emptyset; \emptyset \rangle \parallel \langle \Uparrow (\texttt{input}); \emptyset; \emptyset \rangle$
`LoadController`	$\langle \Uparrow (\texttt{localStats}); \Downarrow (\texttt{centralStats}); \Uparrow (\texttt{output}) \rangle$

Figure E.2: Interaction contracts

Bibliography

Abdelzaher, T. and Bhatti, N. (1999). Web Server QoS Management by Adaptive Content Delivery. In *International Workshop Quality of Service*, IWQoS, London. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=766497.

Abdelzaher, T., Diao, Y., and Hellerstein, J. (2008). Introduction to control theory and its application to computing systems. In *Performance Modeling and Engineering*, pages 185--215. Springer. Link: http://link.springer.com/chapter/10.1007/978-0-387-79361-0_7.

Abdelzaher, T., Shin, K., and Bhatti, N. (2002). Performance guarantees for Web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80--96. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=980028.

Abdelzaher, T. F., Stankovic, J. A., Lu, C., Zhang, R., and Lu, Y. (2003). Feedback performance control in software services. *Control Systems, IEEE*, 23(3):74--90.

Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition.

Ackermann, J. and Turowski, K. (2006). A Library of OCL Specification Patterns for Behavioral Specification of Software Components. In Dubois, E. and Pohl, K., editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 255--269. Springer Berlin / Heidelberg. Link: http://dx.doi.org/10.1007/11767138_18.

Adamczyk, J., Chojnacki, R., Jarząb, M., and Zieliński, K. (2008). Rule Engine Based Lightweight Framework for Adaptive and Autonomic Computing. In *Computational Science – ICCS 2008*. Springer Berlin Heidelberg.

Agha, G. (1990). Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125--141. Link: http://doi.acm.org/10.1145/83880.84528.

Akehurst, D., Bordbar, B., Evans, M., Howells, W., and McDonald-Maier, K. (2006). SiTra: Simple Transformations in Java. In *9th International Conference, MoDELS 2006*, LNCS, Genova.

Akehurst, D. H., Howells, W. G. J., Scheidgen, M., and McDonald-Maier, K. D. (2008). C# 3.0 makes OCL redundant! *ECEASST*, 9.

Andersson, J., Baresi, L., Bencomo, N., De Lemos, R., Gorla, A., Inverardi, P., and Vogel, T. (2012). Software Engineering Processes for Self-adaptive Systems. In de Lemos H. Giese, H. M. M. S., editor, *Software Engineering for Self-adaptive Systems 2*, volume 7475 of *Lecture Notes in Computer Science*. Springer. Link: http://hal.inria.fr/hal-00719003.

Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, G., Kalantar, M., Krishnakumar, S., Pazel, D., Pershing, J., and Rochwerger, B. (2001). Oceano-SLA based management of a computing utility. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, volume 00, pages 855--868. IEEE. Link: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Oc\char"00E9\relaxano+–+SLA+Based+Management+of+a+Computing+Utility#0http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=918085.

Armstrong, J. L., Dacker, B. O., Virding, S. R., and Williams, M. C. (1992). Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services*, pages 157--163.

Arzen, K.-E. and Cervin, A. (2005). Control and embedded computing: Survey of research directions. In *In Proc. 16th IFAC World Congress*, Prague, Czech Republic. Link: http://control.lth.se/documents/2005/arz+05ifac.pdf.

Asadollahi, R. (2009). *StarMX : A Framework for Developing Self-Managing Software Systems*. PhD thesis, University of Waterloo.

Asadollahi, R., Salehie, M., and Tahvildari, L. (2009). StarMX: A framework for developing self-managing Java-based systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 58--67. Ieee. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069074.

Aström, K. (2006). Challenges in control education. In *IFAC Symposium on Advances in Control Education*, Madrid. Link: http://www.quanser.com/english/downloads/toolbox/papers/AstromMadrid2006.pdf.

Aström, K. J. and Murray, R. M. (2009). *Feedback Systems*. Princeton University Press, Princeton, 2.10b edition.

Attariyan, M. and Flinn, J. (2010). Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association.

Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., and van Steen, M. (2005). The Self-Star Vision. In Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., and van Steen, M., editors, *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, page 397. Springer Berlin / Heidelberg. Link: http://dx.doi.org/10.1007/11428589_1.

Baker, H. C. and Hewitt, C. (1977). The incremenatl garbage collection of processes. *ACM SIGART Bulletin*. Link: http://dl.acm.org/citation.cfm?id=806932.

Bencomo, N., Grace, P., Flores, C., Hughes, D., and Blair, G. (2008). Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, ICSE, page 811, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1368088.1368207.

Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., and Consel, C. (2012). DiaSuite: A tool suite to develop Sense/Compute/Control applications. *Science of Computer Programming*, pages 1--28. Link: http://linkinghub.elsevier.com/retrieve/pii/S0167642312000652.

Bézivin, J. and Jouault, F. (2005). Using ATL for checking models. In *Proc. International Workshop on Graph and Model Transformation (GraMoT)*, Tallinn, Estonia. Link: http://www.sciencedirect.com/science/article/pii/S1571066106001393.

Bézivin, J., Jouault, F., Rosenthal, P., and Valduriez, P. (2005). Modeling in the Large and Modeling in the Small. *Model Driven Architecture*, 3599:33--46.

Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run.time. *Computer*, 42(10):22--27. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5280648.

Bouchenak, S., Boyer, F., Hagimont, D., Krakowiak, S., De Palma, N., Quema, V., and Stefani, J.-B. (2005). Architecture-Based Autonomous Repair Management: Application to J2EE Clusters. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 369--370.

Bradley, D., Clair, T. S., Farrellee, M., Guo, Z., Livny, M., Sfiligoi, I., and Tannenbaum, T. (2011). An update on the scalability limits of the Condor batch system. *Journal of Physics: Conference Series*, 331(6):062002. Link: http://stacks.iop.org/1742-6596/331/i=6/a=062002?key=crossref.24c9df79b3d5d797370a310478a768f0http://iopscience.iop.org/1742-6596/331/6/062002.

Bravenboer, M. and Visser, E. (2008). Parse Table Composition. In *First International Conference on Software Language Engineering*.

Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. (2008). Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical report, Christopher Brooks Edward A. Lee Xiaojun Liu Stephen Neuendorffer Yang Zhao Haiyang Zheng Electrical Engineering and Computer Sciences University of California at Berkeley, Berkeley. Link: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA519074.

Brooks, C. X., Lee, E. A., and Tripakis, S. (2010). Exploring models of computation with ptolemy II. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis - CODES/ISSS '10*, page 331, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1878961.1879020http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5751519.

Brooks Jr., F. P. (1987). No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10--19. Link: http://dx.doi.org/10.1109/MC.1987.1663532.

Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. (2011). A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 89--100. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6113791.

Browne, P. (2009). *JBoss Drools Business Rules*. Packt Publishing.

Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., M\\"uller, H., Pezzè, M., and Shaw, M. (2009). Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, pages 48--70. Link: http://www.springerlink.com/index/32NM834361U37368.pdf.

Büttner, F. and Kuhlmann, M. (2008). Problems and Enhancements of the Embedding of OCL into QVT ImperativeOCL. In *Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008*, volume 15 of *OCL*. Link: http://journal.ub.tu-berlin.de/index.php/eceasst/article/viewFile/175/172.

Cabot, J. and Gogolla, M. (2012). Object Constraint Language (OCL): A Definitive Guide. In Bernardo, M., Cortellessa, V., and Pierantonio, A., editors, *12th International School on Formal Methods (SFM)*, volume 7320 of *FSM*, pages 58--90. Springer.

Cámara, J., Canal, C., Cubo, J., and Murillo, J. M. (2007). An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution. In *Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities*, volume 189, pages 21--34. Link: http://linkinghub.elsevier.com/retrieve/pii/S1571066107004884.

Canzanese, R., Kam, M., and Mancoridis, S. (2011). Inoculation against malware infection using kernel-level software sensors. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 101, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1998582.1998600.

Capra, L., Emmerich, W., and Mascolo, C. (2003). CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929--944. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1237173.

Cardelli, L. (1997). *Type Systems, Handbook of Computer Science and Engineering*. CRC Press.

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471--523. Link: http://doi.acm.org/10.1145/6041.6042.

Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., and Mirandola, R. (2009). QoS-driven runtime adaptation of service oriented architectures. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E*, page 131, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1595696.1595718.

Cassou, D., Balland, E., Consel, C., and Lawall, J. (2011). Leveraging software architectures to guide and verify the development of sense/compute/control applications. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 431. Link: http://portal.acm.org/citation.cfm?doid=1985793.1985852.

Cazzola, W. and Poletti, D. (2010). DSL evolution through composition. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, pages 6:1----6:6, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1890683.1890689.

Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A. K., Hanrahan, P., Odersky, M., and Olukotun, K. (2010). Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10. ACM. Link: http://portal.acm.org/citation.cfm?doid=1932682.1869527.

Chen, G. and Kotz, D. (2002). Context aggregation and dissemination in ubiquitous computing systems. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105--114. IEEE Comput. Soc. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1017490http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1017490.

Chen, H. and Hariri, S. (2007). An evaluation scheme of adaptive configuration techniques. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*, page 493, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1321631.1321717.

Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., and Lemos, R. (2009a). Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P., and Magee, J., editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1--26. Springer Berlin / Heidelberg, Berlin, Heidelberg. Link: http://dl.acm.org/citation.cfm?id=1573856.1573859http://dx.doi.org/10.1007/978-3-642-02161-9_1.

Cheng, S.-w. (2008). *Rainbow : Cost-Effective Software*. PhD thesis, Carnegie Mellon University.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2009b). Evaluating the effectiveness of the Rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132--141. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5069082.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2009c). RAIDE for Engineering Architecture-Based Self-Adaptive Systems. Technical report, Computer Science Department, Carnegie Mellon University.

Cisco Systems (2012). Cisco Global Cloud Index: Forecast and Methodology, 2011–2016. Technical report, Cisco Systems.

Clavel, M., Egea, M., and Garcia de Dios, M. A. (2009). Checking unsatisfiability for OCL constraints. In *Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages*. Link: http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/334/319.

Collet, P., Krikava, F., Montagnat, J., Blay-Fornarino, M., and Manset, D. (2010). Issues and Scenarios for Self-Managing Grid Middleware. In *Proceeding of the 2nd workshop on Grids meets autonomic computing*, GMAC, pages 1--10, Washington, USA. ACM. Link: http://dl.acm.org/citation.cfm?id=1809033.

Conan, D., Rouvoy, R., and Seinturier, L. (2007). Scalable processing of context information with COSMOS. In *Proceedings of the 2007 International Conference on Distributed Applications and Interoperable Systems*, pages 210--224. Link: http://www.springerlink.com/index/t15098547n1311g5.pdf.

Correa, A., Werner, C., and Barros, M. (2009). Refactoring to improve the understandability of specifications written in object constraint language. *Software, IET*, 3(2):69--90.

CRA (2003). Association, Computing Research: Final report of the CRA conference on grand research challenges in information systems. Technical report, Computing Reasearch Assocation.

Czarnecki, K. (2005). Overview of Generative Software Development. *Unconventional Programming Paradigms*, pages 326--341. Link: http://link.springer.com/chapter/10.1007/11527800_25.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621--645. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5386627.

David, P.-C., Ledoux, T., Léger, M., and Coupaye, T. (2009). FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of telecommunications - Annales des télécommunications*, 64(1-2).

DMTF (2007). Common Information Model-Simplified Policy Language (CIM-SPL). Technical report, Distributed Management Task Force. Link: www.dmtf.org/standards/publisheddocuments/DSP0231.pdf.

Dobson, S., Denazis, S., and Fernández, A. (2006). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223--259. Link: http://dl.acm.org/citation.cfm?id=1186782.

Dowling, J. and Cahill, V. (2004). Self-managed decentralised systems using K-components and collaborative reinforcement learning. *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems - WOSS '04*, pages 39--43. Link: http://portal.acm.org/citation.cfm?doid=1075405.1075413.

Doyle, J., Francis, B., and Tannenbaum, A. (1992). *Feedback control theory*. Macmillan Publishing Co. Link: http://stuff.mit.edu/afs/athena/course/6/6.241/OldFiles/TA_materials/Spring2011TA/reading/dft.pdf.

Dubochet, G. (2011). *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne.

Eker, J., Janneck, J., Lee, E., Ludvig, J., Neuendorffer, S., and Sachs, S. (2003). Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127--144. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1173203.

Elkhodary, A., Esfahani, N., and Malek, S. (2010). FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, page 7, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1882291.1882296.

Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., and Wallnau, K. (2006). Ultra-Large-Scale Systems -- The Software Challenge of the Future. Technical report, software engineering institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA. Link: www.sei.cmu.edu/library/assets/ULS_Book20062.pdf.

Ferreira da Silva, R., Glatard, T., and Desprez, F. (2012). Self-Healing of Operational Workflow Incidents on Distributed Computing Infrastructures. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 318--325. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6217437.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., and Gjorven, E. (2006). Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62--70. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1605180http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1605180.

Fouquet, F., Barais, O., Plouzeau, N., Jézéquel, J.-M., Morin, B., and Fleurey, F. (2012a). A Dynamic Component Model for Cyber Physical Systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, Bertinoro, Italie. Link: http://hal.inria.fr/hal-00713769.

Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., and Jézéquel, J.-M. (2012b). An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Models 2012*, MODELS, Innsbruck, Autriche. Link: http://hal.inria.fr/hal-00714558.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

France, R., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model Driven development using UML 2.0 : Promises and Pitfalls. *Computer*, 39(2):59--66. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1597089.

France, R. and Rumpe, B. (2007). Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering (FOSE '07)*, number 2 in FOSE, pages 37--54. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4221611.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Ganek, A. G. and Corbi, T. A. (2003). The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5--18. Link: http://dx.doi.org/10.1147/sj.421.0005.

García-Domínguez, A. and Kolovos, D. (2011). EUnit: a unit testing framework for model management tasks. *Model Driven Engineering Languages and Systems*, 6981:395--409. Link: http://link.springer.com/chapter/10.1007/978-3-642-24485-8_29.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46--54. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1350726http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1301377http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1350726.

Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: architectural description of component-based systems. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of component-based systems*, pages 47--67. Cambridge University Press, New York, NY, USA. Link: http://dl.acm.org/citation.cfm?id=336431.336437.

Gat, E. (1998). On three-layer architectures. *Artificial intelligence and mobile robots*. Link: http://www.tu-chemnitz.de/etit/proaut/paperdb/download/gat98.pdf.

Gaudin, B., Vassev, E. I., Nixon, P., and Hinchey, M. (2011). A control theory based approach for self-healing of un-handled runtime exceptions. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, page 217, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1998582.1998633.

George, L., Wider, A., and Scheidgen, M. (2012). Type-Safe Model Transformation Languages as Internal DSLs in Scala. In *Proceeding of the 5th International Conference on Theory and Practice of Model Transformations*, ICMT, Prague. Link: http://www.springerlink.com/index/6766265V42090228.pdf.

Goldsby, H. J., Sawyer, P., Bencomo, N., Cheng, B. H., and Hughes, D. (2008). Goal-Based Modeling of Dynamically Adaptive System Requirements. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*, pages 36--45. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4492385.

Gorton, I., Liu, Y., and Trivedi, N. (2006). An extensible, lightweight architecture for adaptive J2EE applications. *Proceedings of the 6th international workshop on Software engineering and middleware - SEM '06*, page 47. Link: http://portal.acm.org/citation.cfm?doid=1210525.1210537.

Grand, M. (1998). *Patterns in Java, volume 1: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, Inc., New York, NY, USA.

Haller, P. and Odersky, M. (2009). Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202--220. Link: http://dx.doi.org/10.1016/j.tcs.2008.09.019.

Hebig, R., Giese, H., and Becker, B. (2010). Making control loops explicit when architecting self-adaptive systems. In *Proceeding of the second international workshop on Self-organizing architectures - SOAR '10*, SOAR, page 21, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1809036.1809042.

Hellerstein, J. (2010). Why feedback implementations fail: the importance of systematic testing. In *Proceedings of the Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, FeBiD '10, pages 25--26, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1791204.1791209.

Hellerstein, J., Diao, Y., Parekh, S., and Tilbury, D. (2004). *Feedback control of computing systems*. Wiley Online Library. Link: http://onlinelibrary.wiley.com/doi/10.1002/047166880X.fmatter/pdf.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323--364. Link: http://dx.doi.org/10.1016/0004-3702(77)90033-9.

Holzmann, G. J. (2003). *Spin Model Checker*. Addison-Wesley Professional, 1. edition edition.

Horn, P. (2001). Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM. Link: http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

HP (2003). HP Unveils Adaptive Enterprise Strategy to Help Businesses Manage Change and Get More from Their IT Investments.

Huebscher, M. C. and McCann, J. a. (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):1--28. Link: http://portal.acm.org/citation.cfm?doid=1380584.1380585.

Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2 edition edition.

IBM (2006). An Architectural Blueprint for Autonomic Computing. Technical report, IBM. Link: http://people.cs.kuleuven.be/~danny.weyns/csds/IBM06.pdf.

ISO14977 (1996). Information technology -- Syntactic metalanguage -- Extended BNF. Technical report, ISO/IEC. Link: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

Jezequel, J.-M., Combemale, B., Barais, O., Monperrus, M., and Fouquet, F. (2013). Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Journal of Software and Systems Modeling (SoSyM)*.

Johnson, R. and Foote, B. (1988). Designing reusable classes. *Journal of object-oriented programming*, 1(2):1--27. Link: http://www.laputan.org/drc.html.

Jones, T. C. (1978). Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39--63. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5388030.

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS:: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering - GPCE '06*, page 249, New York, New York, USA. ACM Press. Link: http://dl.acm.org/citation.cfm?id=1173744http://portal.acm.org/citation.cfm?doid=1173706.1173744.

Jouault, F. and Kurtev, I. (2006). Transforming Models with ATL. In Bruel, J.-M., editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128--138. Springer Berlin Heidelberg. Link: http://dx.doi.org/10.1007/11663430_14.

Kaczorek, T. (1993). *Linear Control Systems: Synthesis of multivariable systems and multidimensional systems. Vol.2*. INDUSTRIAL CONTROL, COMPUTERS and COMMUNICATION SERIES Series. Research Studies Press Limited. Link: http://books.google.fr/books?id=N81pQgAACAAJ.

Kalayci, S., Dasgupta, G., Fong, L., Ezenwoye, O., and Sadjadi, S. (2010). Distributed and Adaptive Execution of Condor DAGMan Workflows. In *Proceedings of the 2010 International Conference on Software Engineering and Knowledge Engineerin (SEKE'10)*, page 7. Link: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Distributed+and+Adaptive+Execution+of+Condor+DAGMan+Workflows#0.

Kelly, S. (2004). Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In *OOPSLA*. Link: http://www.softmetaware.com/oopsla2004/kelly.pdf.

Kelly, S. and Pohjonen, R. (2009). Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22--29. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5076455.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press. Link: http://www.dsmbook.com/.

Kephart, J. and Chess, D. (2003). The Vision of Autonomic Computing. *Computer*, 36(1):41--50. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1160055http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1160055.

Kephart, J. O. (2011). Autonomic computing: the first decade. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 1--2, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1998582.1998584.

Kolovos, D. (2008). *An extensible platform for specification of integrated languages for model management*. PhD thesis, The University of York. Link: https://www.cs.york.ac.uk/ftpdir/reports/2009/YCST/01/YCST-2009-01.pdf.

Kolovos, D., Paige, R., and Polack, F. (2006). The Epsilon Object Language (EOL). In Rensink, A. and Warmer, J., editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128--142. Springer Berlin / Heidelberg. Link: http://dx.doi.org/10.1007/11787044_11.

Kolovos, D., Paige, R., and Polack, F. (2008a). A framework for composing modular and interoperable model management tasks. In *Model-Driven Tool and Process ….* Link: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.3127.

Kolovos, D., Paige, R., and Polack, F. (2008b). The Epsilon Transformation Language. In *Proceedings of the 2008 International Conference on Model Transformations*, pages 46--60, Zürich, Switzerland. Springer-Verlag. Link: http://www.springerlink.com/index/r575u280706p0371.pdf.

Kolovos, D., Paige, R., and Polack, F. (2009). On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In Abrial, J.-R. and Glässer, U., editors, *Rigorous Methods for Software Construction and Analysis*, volume 5115 of *Lecture Notes in Computer Science*, pages 204--218. Springer Berlin / Heidelberg. Link: http://dx.doi.org/10.1007/978-3-642-11447-2_13.

Krahn, H., Rumpe, B., and Völkel, S. (2010). MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353--372. Link: http://link.springer.com/10.1007/s10009-010-0142-1.

Krikava, F. and Collet, P. (2011). A Reflective Model for Architecting Feedback Control Systems. In *Proceeding of the 2011 International Conference on Software Engineering and Knowledge Engineering*, SEKE, page 7, Miami.

Krikava, F. and Collet, P. (2012a). On the Use of an Internal DSL for Enriching EMF Models. In *Proceedings of the Proceedings of the 2012 International Workshop on OCL and Textual Modelling*, OCL, page 6, Innsbruck. ACM.

Krikava, F. and Collet, P. (2012b). Using Architecture Models to Rapidly Prototype Feedback Control Systems. In *Actes des quatrièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, GDR-GPL, page 2.

Krikava, F., Collet, P., and Blay-Fornarino, M. (2011). Uniform and Model-Driven Engineering of Feedback Control Systems. In *Proceedings of the 2011 International Conference on Autonomic Computing, short paper*, ICAC, page 2, Karlshuhe. ACM Press.

Krikava, F., Collet, P., and France, R. B. (2012). Actor-based Runtime Model of Adaptable Feedback Control Loops. In *Proceeding of the 2012 International Workshop on Models@Runtime in association with MODELS'12*, MRT, page 6, Innsbruck. ACM.

Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K. (2007). Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. *Cluster Computing*, 10(4):443--455. Link: http://link.springer.com/10.1007/s10586-007-0040-9.

Kurtev, I., Bézivin, J., Jouault, F., and Valduriez, P. (2006). Model-based DSL frameworks. In *The 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA'06, page 602, New York, New York, USA. ACM Press. Link: http://dl.acm.org/citation.cfm?id=1176632http://portal.acm.org/citation.cfm?doid=1176617.1176632.

Lapouchnian, A., Yu, Y., Liaskos, S., and Mylopoulos, J. (2006). Requirements-driven design of autonomic application software. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06*, page 7, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1188966.1188976.

Lee, E. A. (2003). Model-Driven Development - From Object-Oriented Design to Actor-Oriented Design. In *Workshop on Software Engineering for Embedded Systems*, Chicago.

Lee, E. A. (2006). The Problem with Threads. *Computer*, 39(5):33--42. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1631937.

Lee, E. A. (2010). Disciplined heterogeneous modeling. In Petriu, D., Rouquette, N., and Haugen, O., editors, *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering, Languages, and Systems (MODELS)*, pages 273--287. Link: http://link.springer.com/chapter/10.1007/978-3-642-16129-2_20.

Lee, E. A., Liu, X., and Neuendorffer, S. (2009). Classes and inheritance in actor-oriented design. *ACM Trans. Embed. Comput. Syst.*, 8(4):29:1----29:26. Link: http://doi.acm.org/10.1145/1550987.1550992.

Léger, M., Ledoux, T., and Coupaye, T. (2010). Reliable dynamic reconfigurations in a reflective component model. *Component-Based Software Engineering*, pages 74--92. Link: http://link.springer.com/chapter/10.1007/978-3-642-13238-4_5.

Leung, J. M.-K., Filiba, T., and Nagpal, V. (2008). VHDL Code Generation in the Ptolemy II Environment. Technical report, University of California, Berkeley.

Lingrand, D., Montagnat, J., and Glatard, T. (2009). Modeling user submission strategies on production grids. In *Proceedings of the 18th ACM international symposium on High performance distributed computing - HPDC '09*, page 121, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1551609.1551633.

Litoiu, M., Woodside, M., and Zheng, T. (2005). Hierarchical model-based autonomic control of software systems. In *Proceedings of the 2005 International Workshop on Design and Evolution of Autonomic Application Software (DEAS '05)*, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1083063.1083071.

Liu, J., Eker, J., Janneck, J., Liu, X., and Lee, E. (2004). Actor-Oriented Control System Design: A Responsible Framework Perspective. *IEEE Transactions on Control Systems Technology*, 12(2):250--262. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281782.

Maggio, M. (2011). *Control Based Design of Computing Systems*. PhD thesis, Politechnico di Milano.

Maggio, M., Hoffmann, H., Santambrogio, M. D., Agarwal, A., and Leva, A. (2011). Decision making in autonomic computing systems. In *Proceedings of the 8th ACM international conference on Autonomic computing - ICAC '11*, ICAC '11, page 201, Karlsruhe, Germany. ACM Press. Link: http://dl.acm.org/citation.cfm?id=1998629http://portal.acm.org/citation.cfm?doid=1998582.1998629.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

McCann, J. and Huebscher, M. (2004). Evaluation issues in autonomic computing. In *Grid and Cooperative Computing-GCC 2004*, pages 597--608. Link: http://link.springer.com/chapter/10.1007/978-3-540-30207-0_74.

McKinley, P., Sadjadi, S., Kasten, E., and Cheng, B. (2004). Composing adaptive software. *Computer*, 37(7):56--64. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1310241.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316--344. Link: http://doi.acm.org/10.1145/1118890.1118892.

Microsoft (2004). Microsoft Dynamic Systems Initiative Overview. Technical report, Microsoft Corporation.

Moors, A., Rompf, T., Haller, P., and Odersky, M. (2012). Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 117--120, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/2103746.2103769.

Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., and Solberg, A. (2009). Models@Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44--51. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5280651.

Mukhija, A. and Glinz, M. (2005). Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of 18th International Conference on Architecture of Computing Systems*, pages 124--138. Link: http://link.springer.com/chapter/10.1007/978-3-540-31967-2_9.

Müller, H., Kienle, H., and Stege, U. (2009). Autonomic computing now you see it, now you don't. *Software Engineering*, pages 32--54. Link: http://link.springer.com/chapter/10.1007/978-3-540-95888-8_2.

Müller, H., Pezzè, M., and Shaw, M. (2008). Visibility of control in adaptive systems. In *Proceedings of the 208 International Workshop on Ultra-Large-Scale Software-Intensive Systems*, ULSSIS, pages 23--26, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1370700.1370707.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 264--278, Berlin, Heidelberg. Springer-Verlag. Link: http://dx.doi.org/10.1007/11557432_19.

Mussbacher, G., Alam, O., Alhaj, M., Ali, S., Am'alio, N., Barn, B., Braek, R., Clark, T., Combemale, B., Cysneiros, L. M., Fatima, U., France, R., Georg, G., Horkoff, J., Kienzle, J., Leite, J. C., Lethbridge, T. C., Luckey, M., Moreira, A., Mutz, F., Oliveira, A. P. A., Petriu, D. C., Sch"ottle, M., Troup, L., and Werneck, V. M. B. (2012). Assessing composition in modeling approaches. In *Proceedings of the Workshop about Comparing Modeling Approaches 2012 (workshop at MODELS 2012)*, CMA'12, New York, NY, USA. ACM. Link: https://dl.acm.org/citation.cfm?id=2459032.

Netcraft (2013). March 2013 Web Server Survey. Technical report, Netcraft. Link: http://news.netcraft.com/archives/2013/03/01/march-2013-web-server-survey.html.

Neti, S. and Muller, H. a. (2007). Quality Criteria and an Analysis Framework for Self-Healing Systems. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, pages 6--6. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4228606.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Tioga Press, Palo Alto. Link: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0934613109.

Nzekwa, R. (2013). *Building Manageable Autonomic Control Loops for Large Scale Systems*. PhD thesis, Universit{é} des Sciences et Technologie de Lille - Lille I. Link: http://tel.archives-ouvertes.fr/tel-00843874.

OASIS (2007). Service Component Architecture (SCA). Technical report, OASIS Open CSA. Link: http://www.oasis-opencsa.org/sca.

Object Management Group (2008). MOF Model to Text Transformation Language (MOFM2T). Technical report, Object Management Group.

Object Management Group (2011a). MOF™ Query / View / Transformation (QVT). Technical report, Object Management Group. Link: http://www.omg.org/spec/QVT/1.1/.

Object Management Group (2011b). Unified Modeling Language™ (UML®). Technical report, Object Management Group. Link: http://www.omg.org/spec/UML/2.4.1/.

Object Management Group (2012a). Common Object Request Broker Architecture (CORBA)®. Technical report, OMG. Link: http://www.omg.org/spec/CORBA/.

Object Management Group (2012b). OMG Object Constraint Language (OCL). Technical report, Object Management Group. Link: http://www.omg.org/spec/OCL/2.3.1.

Object Management Group (2012c). Systems Modeling Language (SysML). Technical report, Object Management Group. Link: http://www.omg.org/spec/SysML/1.3/.

Odersky, M. (2011). The Scala Language Specification. Technical report, Programming Methods Laboratory, EPFL. Link: www.scala-lang.org/docu/files/ScalaReference.pdf.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An Overview of the Scala Programming Language. Technical report, École Polytechnique Fédérale de Lausanne.

Ohloh (2013). Ohloh public directory of Free and Open Source Software. Link: http://www.ohloh.net/.

Oliveira, B., Moors, A., and Odersky, M. (2010). Type classes as objects and implicits. *ACM Sigplan Notices*, 45(10):341. Link: http://dl.acm.org/citation.cfm?id=1869489.

Oreizy, P., Rosenblum, D. S., and Taylor, R. N. (1998). On the Role of Connectors in Modeling and Implementing Software Architectures. Technical report, DEPARTMENT OF INFORMATION AND COMPUTER SCIENCE, UNIVERSITY OF CALIFORNIA.

Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N., and a.C. Polack, F. (2009). The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 162--171. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5090524.

Patikirikorala, T., Colman, A., Han, J., and Wang, L. (2012). A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 33--42.

Pfeiffer, R. and Wasowski, A. (2012). Cross-language support mechanisms significantly aid software development. In *In the proceedings of the 2012 Model Driven Engineering Languages and System*, MODELS, pages 168--184, Innsbruck, Austria. Link: http://link.springer.com/chapter/10.1007/978-3-642-33666-9_12.

Ramaprasad, A. (1983). On the definition of feedback. *Behavioral Science*, 28(1):4--13. Link: http://dx.doi.org/10.1002/bs.3830280103.

Ramirez, A. J. and Cheng, B. H. C. (2010). Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 49--58, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1808984.1808990.

Romero, D., Rouvoy, R., and Chabridon, S. (2010). Middleware Approach for Ubiquitous Environments. *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, pages 113--135.

Rompf, T. and Odersky, M. (2010). Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127--136, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1868294.1868314.

Rompf, T., Sujeeth, A. K., Amin, N., Brown, K. J., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., and Odersky, M. (2013). Optimizing data structures in high-level programs. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 497, New York, New York, USA. ACM Press. Link: http://dl.acm.org/citation.cfm?doid=2429069.2429128.

Rose, L. M., Matragkas, N., Kolovos, D. S., and Paige, R. F. (2012). A feature model for model-to-text transformation languages. *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, pages 57--63. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6226015.

Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. (2008). The Epsilon Generation Language. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA, pages 1--16, Berlin, Germany. Springer-Verlag. Link: http://www.springerlink.com/index/h182m536v0471545.pdf.

Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., and Scholz, U. (2008). MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In *MobMid '08: Proceedings of the 1st workshop on Mobile middleware*, pages 164--182, New York, NY, USA. ACM.

Sadjadi, S. and McKinley, P. (2004). ACT: an adaptive CORBA template to support unanticipated adaptation. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 74--83. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281570.

Sadjadi, S. M., McKinley, P. K., Cheng, B. H., and Stirewalt, R. K. (2004). TRAP/J: Transparent Generation of Adaptable Java Programs. *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, 3291. Link: http://link.springer.com/chapter/10.1007/978-3-540-30469-2_28.

Salehie, M. and Tahvildari, L. (2005). Autonomic computing: emerging trends and open problems. *ACM SIGSOFT Software Engineering Notes*. Link: http://dl.acm.org/citation.cfm?id=1083082.

Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1--42. Link: http://portal.acm.org/citation.cfm?id=1516538.

Sánchez Cuadrado, J., Canovas, J., and Garcia Molina, J. (2012). Comparison between internal and external DSLs via RubyTL and Gra2MoL. In Mernik, M., editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global. Link: http://hal.inria.fr/hal-00752687.

Sawyer, P., Bencomo, N., Hughes, D., Grace, P., Goldsby, H. J., and Cheng, B. H. C. (2007). Visualizing the Analysis of Dynamically Adaptive Systems Using i* and DSLs. In *Second International Workshop on Requirements Engineering Visualization (REV 2007)*, number Rev, pages 3--3. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4473003.

Schmidt, D. C. (2002). Middleware for real-time and embedded systems. *Communications of the ACM*, 45(6). Link: http://portal.acm.org/citation.cfm?doid=508448.508472.

Schmidt, D. C. (2006). Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39(2):25--31.

Seinturier, L., Merle, P., Furnier, D., Dolet, N., Schiavoni, V., and Stefani, J.-B. (2009). Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceeding of the 6th IEEE International Conference on Service Computing*, SCC, pages 268--275, Bangalore, India. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5283947.

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):1--12. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1231150.

Shaw, M. (1995). Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1):38. Link: http://portal.acm.org/citation.cfm?id=225911.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. (2012). Making middleboxes someone else's problem. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication - SIGCOMM '12*, page 13, New York, New York, USA. ACM Press. Link: http://dl.acm.org/citation.cfm?doid=2342356.2342359.

Siek, J. G. (2010). General purpose languages should be metalanguages. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 3--4, New York, NY, USA. ACM. Link: http://doi.acm.org/10.1145/1706356.1706358.

Steele, G. L. (1999). Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221--236.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional. Link: http://www.eclipse.org/emf.

Sun, M. (1997). Java Code Conventions. Technical report, Sun Microsystems. Link: http://www.oracle.com/technetwork/java/codeconv-138413.html.

Sutter, H. and Larus, J. (2005). Software and the Concurrency Revolution. *ACM Queue*, 3(7):54--62. Link: http://doi.acm.org/10.1145/1095408.1095421.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Taha, W. and Sheard, T. (1997). Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, Amsterdam, The Netherlands.

Tamura, G., Casallas, R., Cleve, A., and Duchien, L. (2010). QoS Contract-Aware Recon guration of Component Architectures Using E-Graphs. In *7th International Workshop on Formal Aspects of Component Software*, number i in FACS, page 2010.

Tamura, G., Villegas, N., Müller, H., A., H., Duchien, L., and Seinturier, L. (2013). Improving context-awareness in self-adaptation using the DYNAMICO reference model. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, number i in SEAMS, pages 153--162. Link: http://dl.acm.org/citation.cfm?id=2487361.

Taylor, R. N., Medvidovic, N., and Dashofy, E. (2010). *Software Architecture: Foundations, Theory, and Practice*. Wiley. Link: http://books.google.fr/books?id=j9pdGQAACAAJ.

Tewari, V. and Milekovic, M. (2006). Standards for Autonomic Computing. *Intel Technology Journal*, 10(4).

Thain, D., Tannenbaum, T., and Livny, M. (2005). Distributed computing in practice: The Condor experience. *Concurrency and Computation Practice and Experience*, 17(2-4):323--356. Link: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.6.3035&rep=rep1&type=pdf.

Tolvanen, J. and Kelly, S. (2005). Defining domain-specific modeling languages to automate product derivation: Collected experiences. *Software Product Lines*, pages 198--209. Link: http://link.springer.com/chapter/10.1007/11554844_22.

Tolvanen, J.-P. (2012). More or less languages: panel at MODELS 2012. Link: http://www.metacase.com/blogs/jpt/blogView?showComments=true&entry=3531392326.

Typesafe (2013). Akka 2.1.2 Documentation. Technical report, Typesafe. Link: http://doc.akka.io/docs/akka/2.1.2/Akka.pdf.

van Vliet, H. (2008). *Software Engineering: Principles and Practice*. John Wiley & Sons, 3rd editio edition.

Villegas, N., Tamura, G., Müller, H., Duchien, L., and Casallas, R. (2013). DYNAMICO : A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. *Software Engineering for Self-adaptive Systems 2*, pages 265--293. Link: http://link.springer.com/chapter/10.1007/978-3-642-35813-5_11.

Villegas, N. M., Müller, H. A., Tamura, G., Duchien, L., and Casallas, R. (2011). 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*, volume 1 of *SEAM '11*, page 80, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1988008.1988020.

Voelter, M. (2011). Language and IDE Modularization and Composition with MPS. In *4th Generative & Transformational Techniques in Software Engineering*, Braga.

Voelter, M., Ratiu, D., Schaetz, B., and Kolb, B. (2012). mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12. Link: http://dl.acm.org/citation.cfm?id=2384767.

Vogel, T. and Giese, H. (2010). Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '10*, SEAMS, pages 39--48, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1808984.1808989.

Vogel, T. and Giese, H. (2012). A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, number 3 in SEAMS, pages 129--138. IEEE. Link: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6224399.

Vromant, P., Weyns, D., Malek, S., and Andersson, J. (2011). On interacting control loops in self-adaptive systems. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, SEAMS, New York, New York, USA. ACM Press. Link: http://portal.acm.org/citation.cfm?doid=1988008.1988037.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1994). A Note on Distributed Computing. Technical report, Sun Microsystems. Link: http://link.springer.com/content/pdf/10.1007/3-540-62852-5_6.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language, Second Edition*. Addison-Wesley.

Weyns, D., Iftikhar, M., Malek, S., and Andresson, J. (2012). Claims and supporting evidence for self-adaptive systems: A literature study. In *Proceeding of the 2012 International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Link: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6224395.

Weyns, D. and Malek, S. (2010). FORMS: a formal reference model for self-adaptation. In *Proceedings of the 2010 International Conference on Autonomic Computing (ICAC'10)*, pages 205--214. Link: http://portal.acm.org/citation.cfm?id=1809078.

White, J., Schmidt, D., and Gokhale, A. (2005). Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In *Proc. 8th Intl. Conf. Model Driven Engineering Languages and Systems*. Link: http://link.springer.com/chapter/10.1007/11557432_45.

Wider, A. (2011). Towards combinators for bidirectional model transformations in scala. *Software Language Engineering*. Link: http://www.springerlink.com/index/V5771U562822322N.pdf.

Wilke, C. and Demuth, B. (2010). UML is still inconsistent ! How to improve OCL Constraints in the UML 2.3 Superstructure. In *Workshop on OCL and Textual Modelling*, volume 36 of *OCL*, page 19.

Willink, E. (2011a). Modeling the OCL Standard Library. In *Workshop on OCL and Textual Modelling*, OCL, page 20.

Willink, E. D. (2011b). Re-engineering Eclipse MDT / OCL for Xtext. In *2011 Workshop on OCL and Textual Modelling*, pages 1--15. Link: http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/444.

Willink, E. D. (2012). An extensible OCL virtual machine and code generator. *Proceedings of the 12th Workshop on OCL and Textual Modelling - OCL '12*, pages 13--18. Link: http://dl.acm.org/citation.cfm?doid=2428516.2428519.

Wyk, E. V., Krishnan, L., Bodin, D., and Schwerdfeger, A. (2007). Attribute grammar-based language extensions for Java. In *ECOOP 2007*. Link: http://link.springer.com/chapter/10.1007/978-3-540-73589-2_27.

Xiong, Y. and Lee, E. (2000). An Extensible Type System for Component-Based Design. In *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Berlin, Germany.

Zhang, J. and Cheng, B. H. C. (2006). Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, ICSE '06, page 371, New York, New York, USA. ACM Press. Link: http://doi.acm.org/10.1145/1134285.1134337http://portal.acm.org/citation.cfm?doid=1134285.1134337.

Zhao, Y. (2003). A Model of Computation with Push and Pull Processing. Technical report, Technical Memorandum UCB/ERL M03/51, University of California, Berkeley.

Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., and Ernst, M. D. (2007). Object and reference immutability using java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 75, New York, New York, USA. ACM Press. Link: http://dl.acm.org/citation.cfm?id=1287624.1287637.

Domain-Specific Modeling Language for Self-Adaptive Software System Architectures

The vision of Autonomic Computing and Self-Adaptive Software Systems aims at realizing software that autonomously manage itself in presence of varying environmental conditions. Feedback Control Loops (FCL) provide generic mechanisms for self-adaptation, however, incorporating them into software systems raises many challenges.

The first part of this thesis addresses the integration challenge, *i.e.*, forming the architecture connection between the underlying adaptable software and the adaptation engine. We propose a domain-specific modeling language, FCDL, for integrating adaptation mechanisms into software systems through external FCLs. It raises the level of abstraction, making FCLs amenable to automated analysis and implementation code synthesis. The language supports composition, distribution and reflection thereby enabling coordination and composition of multiple distributed FCLs. Its use is facilitated by a modeling environment, Actress, that provides support for modeling, verification and complete code generation. The suitability of our approach is illustrated on three real-world adaptation scenarios.

The second part of this thesis focuses on model manipulation as the underlying facility for implementing Actress. We propose an internal Domain-Specific Language (DSL) approach whereby Scala is used to implement a family of DSLs, Sigma, for model consistency checking and model transformations. The DSLs have similar expressiveness and features to existing approaches, while leveraging Scala versatility, performance and tool support.

To conclude this thesis we discuss further work and further research directions for MDE applications to self-adaptive software systems.

Langage de Modélisation Spécifique au Domaine pour les Architectures Logicielles Auto-Adaptatives

Le calcul autonome vise à concevoir des logiciels qui prennent en compte les variations dans leur environnement d'exécution. Les boucles de rétro-action (FCL) fournissent un mécanisme d'auto-adaptation générique, mais leur intégration dans des systèmes logiciels soulève de nombreux défis.

Cette thèse s'attaque au défi d'intégration, c.à.d. la composition de l'architecture de connexion reliant le système logiciel adaptable au moteur d'adaptation. Nous proposons pour cela le langage de modélisation spécifique au domaine FCDL. Il élève le niveau d'abstraction des FCLs, permettant l'analyse automatique et la synthèse du code. Ce langage est capable de composition, de distribution et de réflexivité, permettant la coordination de plusieurs boucles de rétro-action distribuées et utilisant des mécanismes de contrôle variés. Son utilisation est facilitée par l'environnement de modélisation Actress qui permet la modélisation, la vérification et la génération du code. La pertinence de notre approche est illustrée à travers trois scénarios d'adaptation réels construits de bout en bout.

Nous considérons ensuite la manipulation de modèles comme moyen d'implanter Actress. Nous proposons un Langage Spécifique au Domaine interne qui utilise Scala pour implanter une famille de DSLs. Il permet la vérification de cohérence et les transformations de modèles. Les DSLs résultant ont des propriétés similaires aux approches existantes, mais bénéficient en plus de la souplesse, de la performance et de l'outillage associés à Scala.

Nous concluons avec des pistes de recherche découlant de l'application de l'IDM au domaine du calcul autonome.