A Framework for Generating and Deploying Dynamic Performance Monitors for Self-Adaptive Software Systems

Master Thesis

Presented in partial fulfillment to obtain the Title of
Magister in Informatics and Telecommunications

by

Miguel Jiménez

Advisor: Gabriel Tamura, PhD

Department of Information and Communication Technologies
Faculty of Engineering

July 15, 2016

Contents

List of Tables

List of Figures

Abstract

Software systems are pervasively becoming part of users' daily life. In face of increasingly competitive markets, companies are concerned with continuous service delivery and accomplishment of agreed levels of fulfillment in service performance. Moreover, advances in autonomic computing for strengthening responsiveness and resilience of service delivery have promoted the design of re-configurable systems able to modify their structure and behavior at runtime. Guaranteeing specific levels of performance in dynamic systems implies measurement mechanisms to evaluate metrics that must be updated periodically, following the evolution of the system's requirements and also of its environment. Therefore, to actually ensure service performance, system administrators require monitoring infrastructures to continuously measure the satisfaction of the different system's performance factors capable of (i) dynamically updating its monitoring strategies as the managed system's requirements or the environment evolves; (ii) realizing the deployment and integration of monitoring components at runtime; and (iii) providing the means to generate composable, traceable, and controllable monitoring capabilities. In this Thesis, we address these challenges with the design of a scalable dynamic monitoring architecture, which implements and resolves dynamic monitoring concerns in context-aware autonomic systems. Based on this architecture, we also design and implement two domain-specific languages, Pascani and Amelia, that facilitate the development of the aforementioned monitors, suitable to be integrated in the architecture, and to automate their deployment into the running infrastructure of the target system. The relevance of our work lies in the realization of automated mechanisms to support the preservation of the monitoring pertinence in face of highly changing environments.

Keywords: Performance Monitoring, Dynamic Monitoring, System Deployment, Self-Adaptive Software Systems

Resumen

Los sistemas de software están cada vez más compenetrados en la vida cotidiana de los usuarios. Ante mercados cada vez más competitivos, las empresas se preocupan por la prestación continua de servicios de software, y por el cumplimiento de niveles de rendimiento acordados respecto a dichos servicios. Además, avances en computación autonómica para el fortalecimiento de la capacidad de respuesta y recuperación en la prestación de servicios ha promovido el diseño de sistemas reconfigurables capaces de modificar su estructura en tiempo de ejecución. Garantizar niveles específicos de rendimiento en sistemas dinámicos implica disponer de mecanismos de medición para evaluar métricas que deben ser actualizadas periódicamente, siguiendo la evolución de los requerimientos del sistema y también su entorno. Por lo tanto, para asegurar realmente el rendimiento de los servicios, los administradores de sistemas requieren infraestructuras de monitoreo para medir continuamente la satisfacción de los diferentes factores de rendimiento capaces de (i) actualizar dinámicamente sus estrategias de monitoreo en la medida que los requerimientos del sistema o su entorno evolucionen; (ii) realizar el despliegue y la integración de los componentes de monitoreo en tiempo de ejecución; y (iii) proveer los medios para generar funcionalidades de monitoreo componibles, rastreables y controlables. En esta tesis dichos retos se abordan con el diseño de una arquitectura de monitoreo dinámica y escalable, que implementa y resuelve preocupaciones de monitoreo dinámico en sistemas autonómicos sensibles al contexto. Con base en esta arquitectura, también se diseñan e implementan dos lenguajes de dominio específico, Pascani y Amelia, que facilitan el desarrollo de los monitores mencionados anteriormente, adecuados para ser integrados en la arquitectura y para automatizar su despliegue en la infraestructura del sistema objetivo mientras este está operando. La relevancia del trabajo presentado radica en la realización de mecanismos para soportar la preservación de la pertinencia del monitoreo, frente a entornos altamente dinámicos.

Palabras Clave: Monitoreo de Rendimiento, Monitoreo Dinámico, Despliegue de Sistemas, Sistemas de Software Autoadaptativo

dedicated to my parents,
who taught me the language of love and forgiveness.

Acknowledgments

I would like to express my deep gratitude to my thesis advisor Dr. Gabriel Tamura for all the comments, guidance and critical analysis throughout this process. Although there is still a long road ahead, I found myself being a very different person now.

I would like to thank Dr. Norha Villegas, who read over drafts and made suggestions to improve this document. She also gave me positive comments that encouraged me to continue working hard and improving everyday.

To all my friends and mates from the i2t/DRISO research team I can only offer my most sincere thanks, for all the support and feedback. I spent very joyful moments with all of you that I hope we can repeat sometime.

Finally, I would like to thank the Faculty of Engineering of Universidad Icesi for giving me a full tuition scholarship for this master.

Chapter 1

Introduction

1.1 Context and Motivation

Software systems have become a fundamental support for everyday activities, in both personal and business contexts. On the one hand, individuals rely on software to accomplish duties and responsibilities across their lifestyle; multimedia applications have become part of their daily habits, including social networks, messaging, and online streaming of music, movies, and TV shows. On the other hand, companies are increasingly dependent on software technology to support their business objectives. Software is not only a base tool to cope with tedious administrative tasks, but also the personal and business infrastructure to deliver value-added services to customers. These two perspectives show the increasing dependence of users on their software applications and, as a result, their increasing quality expectations on the provided application services (*e.g.*, users want to see video streaming in one smooth playback). Thus, in order to promote business strengthening, stakeholders are concerned with the fulfillment of certain quality attributes (QA), especially those sensibly impacting the system behavior as perceived by the client.

To ensure the fulfillment of the system's QAs (*e.g.*, guaranteeing service level agreements), software designers have to consider mechanisms to measure the clients satisfaction, and to overcome foreseeable disturbances able to deviate the system from its expected behavior. These disturbances may be identified either in early stages of development, enabling architects to address them from a design perspective, or in the operational stage, leaving system administrators with the responsibility of overcoming them. Effective assurance can then be achieved by accurately identifying the root causes of the detected problems, and generating adequate counter-effects to correct the system behavior. Delegating such responsibility to system administrators implies IT staff reacting to changes in the system behavior, at runtime. This can lead into ineffective assurance of the quality contracts, as the identification and response processes performed by humans can delay the corresponding mitigating actions. Instead, QAs assurance at runtime should rely on

systematic procedures; these procedures should be supported by the realization of monitoring and analysis concerns, such as mechanisms to measure, monitor, and control the system behavior [23].

As a result, a critical task to continuously fulfill the system QAs is to measure and monitor the system behavior. Nonetheless, as monitoring is not generally considered as a first class entity in the traditional software engineering process, the measurement of relevant monitoring variables, related to the system QAs, is commonly performed by manually developing and adding measurement code into several locations in the application source code, resulting in entangled low-level implementations [23]. Despite this approach is simple at the beginning, it has multiple drawbacks when the application components are generated automatically. First, after the system components to be measured are generated, and then manually modified with the measurement code, the generation mechanism cannot be used anymore, as the manually inserted code would be lost. Second, as measurement variables are not defined as part of a standard measurement mechanism, they are difficult to locate and share between developers [23]. Third, manually inserted code for measurement targets the particular subsystem, making it not reusable.

Moreover, beyond taking raw measurement data, there should be a monitoring logic in charge of processing and composing monitoring data in the form of variables and context events, to report valuable data to appropriate stakeholders, such as service response time to system administrators, and error occurrence to product managers. In practice, two monitoring considerations arise: first, once a problem has been discovered, such as an unexpected long service response time or high memory consumption in a short period of time, how to identify the corresponding system components causing the problem? Reported data must contain the necessary information to decompose the monitored measurements, allowing to dig further into the reported data. Second, quality scenarios are subject to change, either because they are re-negotiated or because new scenarios become relevant. Hence, manually developed measurement code must be carefully analyzed and updated by application developers, given that they offer no support for self-adaptive monitoring strategies to address changes in monitoring requirements [42].

Developing cost-effective monitoring solutions requires more than functional mechanisms, it requires mechanisms providing standard and adequate specification formats, with the appropriate level of abstraction and expressiveness. These characteristics configure the proper scenario to consider Domain Specific Languages (DSLs) as a suitable alternative for automating the generation of monitoring components systematically. DSLs can increase flexibility and reliability, thus increasing productivity [29]. Additionally, DSLs can reduce considerable efforts in composing monitoring specifications and measurements. Such solutions should also support runtime operations, that is, operations to be applied while the system is in execution. These operations include managing custom measurements and parameters, modifying the sampling frequency of discrete measurements,

and controlling the monitoring mechanism itself. In order to effectively apply these operations, system administrators must be able to automatically and reliably deploy the necessary monitoring infrastructure to continuously gather relevant information from the system. For these runtime operations to be reliable, the generated components must be capable to report relevant traces of the execution during regular operation, including before and after system updates take place.

Although system administrators are continuously observing the system behavior, context conditions change dramatically in short periods of time, leaving them with no opportunity to react properly and timely. In the mean time, interruptions in service provisioning take place, and therefore, as quality expectations are not met, customer loyalty decreases. Furthermore, as systems continuously grow and evolve into distributed networks of components, effective administration and service delivery become a challenging problem [11]. In the social network domain, for instance, Twitter experienced rough times when their infrastructure went over capacity; between 2008 and 2012 they reported 12 severe service outages[1] before they completely change their infrastructure in 2013[2]. Some of these outages and response time increments were caused by unexpected service requests during the 2010 FIFA World Cup [44] and the 2012 Summer Olympics[3]. To overcome these situations, administrative tasks have to be automated, abstracting IT staff's knowledge into automated mechanisms able to properly respond to context changes. By doing this, responsiveness and resilience of service delivery are strengthened [11]. These tasks have been grouped into self-managing capabilities, as an approach to realize the vision of Autonomic Computing. That is, systems that manage themselves in accordance to high level objectives specified by administrators [28].

In spite of self-management capabilities reducing human intervention in system administration, developing self-awareness mechanisms (*i.e.*, mechanisms enabling systems with awareness regarding their own behavior) requires monitoring mechanisms able to dynamically update their measurement strategy. That is, measurement procedures should be removable from the already monitored components, and deployable on new components to start monitor them. Similarly, it must be possible to modify the set of monitored variables by adding new ones or deleting existing ones. As a result, it must be possible to substitute the monitoring logic dynamically at runtime, as analyzed in [43, 40]. A complete or partial renewal of the monitoring infrastructure requires performing deployment actions on monitors, that is: compiling new monitor implementations, transporting them to the corresponding computing nodes, executing them, and resolving their binding dependencies [20]. Furthermore, previous versions of them might have to be removed, and some parts of the system might have to be recompiled into the running infrastructure. As in

[1] https://en.wikipedia.org/wiki/Twitter#Outages

[2] http://www.wired.com/2013/11/qa-with-chris-fry/

[3] http://www.theguardian.com/technology/2012/jul/26/twitter-down-olympics

the monitoring case, according to the separation of concerns principle [17, 15], these deployment actions require tailored specification formats, and adequate power of expression.

In summary, the motivation behind this thesis is the need for providing software systems with monitoring infrastructures generated automatically and deployed at runtime. These dynamic monitoring infrastructures are required to continuously guarantee quality attributes in software applications that face changing context conditions that can violate these attributes' fulfillment at runtime. In spite of existing proposals focusing on the software engineering for self-adaptive software systems, such as the DYNAMICO reference model, the aforementioned challenges are still open [43].

1.2 Problem Statement

Considering the motivation presented in the previous section, the problem statement of this thesis is stated as follows:

> *Given an arbitrary service-component based software application, and a performance monitoring specification, automatically generate the monitor(s) for this quality attribute, and deploy and integrate the generated monitors into the software application, at runtime. These monitors' implementation must conform to the Service-Component Architecture (SCA) specification, and provide the functionalities of the monitoring element of the DYNAMICO reference model [43] (i.e., provide dynamic performance measurement mechanisms, deployable through reconfiguration actions at runtime).*

1.3 Challenges

As previously mentioned in Section 1.1, monitoring mechanisms are key elements in ensuring quality attributes. Despite monitoring has been widely applied by practitioners through several approaches, the need for resilient systems to face today's requirements has raised new challenges. In order to cope with these still unresolved monitoring challenges, self-adaptive software systems require dynamic monitoring infrastructures to continuously measure the satisfaction of the system's quality attributes, beyond manually inserted code, capable of:

1. dynamically updating its monitoring strategies as the managed system's requirements or the environment evolve.
2. realizing the deployment and integration of monitoring components at runtime.
3. providing composable, traceable, and controllable monitoring capabilities.
4. reporting composed and calculated monitoring data, with the associated raw measurements.

1.4 Research Objectives

1.4.1 General

To develop standard mechanisms that generate monitoring infrastructures for performance factors (e.g., latency and throughput), capable of deploying different measurement strategies at runtime, in order to continuously satisfy performance service level indicators in service-component based software systems.

1.4.2 Specific

1. To design a software reference architecture for the monitoring infrastructure required to supervise the satisfaction of the performance quality attribute in component-based software systems, with well defined and standard communication protocols.
2. To develop standard mechanisms that generate composable and traceable monitoring components with the corresponding measurement procedures, from the specification of monitoring concerns regarding the performance quality attribute.
3. To design and develop standard mechanisms to deploy software components, including the preparation of target assets, their transportation to (possibly) distributed computing nodes, their execution, and system resources' clean up.
4. To design and develop a strategy for integrating measurement strategies by deploying monitoring artifacts at runtime, and updating the target system's running infrastructure.

1.5 Methodology

The adopted methodology for developing this thesis is a qualitative approach [12] to both develop and validate our solution. On the one hand, in the development phase, qualitative methods are used to guide our state of the art exploration, starting with a literature review mainly focused on the motivation and need for self-managed software systems, as a way to advance on the design of autonomic systems [28]. Moreover, since our interest is to automate monitoring (*i.e.*, develop the means to realize self-awareness), we require reference models and architectures to effectively implement dynamic monitoring infrastructures through self-adaptation mechanisms. Additionally, we also need to survey mechanisms for specifying and deploying monitoring software components, using a Domain Specific Language (DSL) approach. On the other hand, in the evaluation phase, we use qualitative methods to evaluate *i)* the completeness of our solution regarding the requirements and related quality scenarios presented in chapter **??**, and *ii)* the expressiveness and usability of the two DSLs composing our solution. To this end, we perform a workshop for each language, using a non-trivial and relevant case study.

In order to accomplish the established set of specific objectives, we achieved the following milestones in this project:

1. Elicitation of requirements and architectural design.
   (a) Identification and specification of the monitoring requirements to realize performance quality self-awareness.
   (b) Specification of the deployment requirements to automate component distribution and execution, and service binding.
   (c) Design of the software architecture for the envisioned monitoring infrastructure.
   (d) Development of standard mechanisms to supply software components with capabilities for traceability and logging.
2. Design and implementation of two DSLs to specify, generate, compile, and execute performance monitoring concerns, as well as specify and realize deployment updates at runtime.
   (a) Design of each DSL corresponding syntax and semantics.
   (b) Design of the context-free grammars in correspondence with the proposed syntax.
   (c) Design and implementation of the translation and execution models according to the proposed semantics.
3. Development of a proof-of-concept implementation, including the case study artifacts, the corresponding quality attribute monitors and deployment descriptors using the developed DSLs.
4. Analysis and evaluation of results.

1.6 Thesis Organization

The remaining chapters of this thesis are organized as follows. Chapter 2 presents the background and state of the art of the main concepts involved in this research: Component Based Software Engineering, Service Component Architecture, Self-managed Software Systems, Domain Specific Languages, and most importantly, System Monitoring and Software Deployment. Chapter 3 analyzes the high-level functional requirements for designing and implementing our envisioned dynamic monitoring architecture, along with the related quality considerations. Chapter 4 presents Pascani and Amelia, two domain-specific languages for specifying dynamic performance monitors, and deploying them into the target system's running infrastructure, respectively. Chapter 5 details the design and realization of our proof-of-concept implementation. Lastly, Chapter 6 concludes this thesis and proposes future work.

1.7 Contribution Summary

The main contributions of this thesis are:

An Architecture for Dynamic Performance Monitoring

We present an scalable dynamic monitoring architecture that implements and resolves dynamic monitoring concerns in context-aware self-adaptive software systems. Our architecture promotes the generation of composable, traceable, and controllable monitoring components. The relevance of this contribution lies in the realization of automated mechanisms to support the preservation of the monitoring pertinence in face of highly changing environments. This work has been partially presented in [26, 3, 4].

A Performance Monitoring Language

We created a domain-specific language named Pascani to ensure that monitoring concerns can be specified using a standard and adequate format, with the appropriate level of abstraction. Pascani allows to define and update context variables in models that can be shared across several monitor specifications. From the monitoring specifications, the language engine generates SCA-compliant components that are composable, traceable, and controllable. This work has been partially presented in [4].

A Deployment Language for Component-based Systems

We created a domain-specific language named Amelia to abstract and facilitate the systematic realization of system deployment. The language constructs offer great power of expression for building systems and executing artifacts into distributed computing infrastructures. From Amelia specifications, the language engine generates an executable Java program with status reporting and logging capabilities able to deploy SCA-compliant components to the specified computing nodes.

Chapter 2

Background and State of the Art

Section 1.1 established our motivation to advance in the automated generation of monitoring systems as a required medium to fulfill system performance assurances. In order to understand the relevance of this thesis, and its associated challenges listed in Section 1.3, it is important to establish the background and the main concepts involved in this project: Component Based Software Engineering (CBSE), Service Component Architecture (SCA), Self-managed Software Systems, Domain Specific Languages (DSL), and most importantly, System Monitoring and Software Deployment. The order in which we introduce such concepts follows a general-to-specific sequence.

2.1 Component-based Software Engineering

Component-based Software Engineering (CBSE) is an approach to software development that relies in software reuse, based on a set of design principles and standards for implementation, documentation, and deployment, encapsulated into a component model. In this paradigm, components are opaque software units with well defined services and explicit dependencies on other components. Services are made visible through interfaces, making it possible for components to rely on expected behavior, and therefore, be developed and deployed independently [5]. Interaction among components is realized by communicating components, binding required services (*i.e.*, dependencies) with provided services, through communication protocols (*e.g.*, SOAP, REST or RMI).

According to the CBSE vision, components are independent and flexible units of software that allow to build systems with improved predictability based on components' properties, component markets, and reduced time-to-market [5]. However, one of the open challenges in CBSE is Component trustworthiness, that is, how can components from unknown sources be trusted? and in the same sense, who will certify the quality of those components? These and other concerns have been addressed by the software community, producing new component models (*e.g.*, Service Component Architecture [7]), middleware (*e.g.*, FraSCAti[37]), and related technologies.

The Service Component Architecture Specification.

Extending the vision of CBSE, the SCA specification defines a general approach to assemble Enterprise Applications (EA) based on components and *services*, providing a standardized mechanism for wrapping individual services into high-level business pieces [10, 38]. Accordingly, since SCA follows the vision of Service Oriented Architecture (SOA), by supporting these high-level components, SCA not only standardizes but also simplifies the build, deployment, and management of EAs.

High-level components are realized through `composites`, that is, XML descriptors containing components and service dependencies, including both provided and required services. These inner components in a SCA application may be implemented using different programming languages and/or technologies, such as Java, C++, OSGi, and BPEL, and wired using different binding implementations, such as Web Services, JMS, RMI, and JCA.

Current implementations of the SCA specification include open source projects, such as Apache Tuscany [2], Fabric3 [33], and FraSCAti [37], and commercial products, such as IBM Websphere Server Feature Pack for SCA [25], and Oracle Tuxedo [36]. Nonetheless, FraSCAti is the only implementation that currently offer the capabilities of dynamic reconfiguration at runtime. The relevance of CBSE and SCA in this thesis is that they define the target platforms for our contribution (*i.e.*, code generation).

2.2 Self-managed Software Systems

In face of increasing levels of complexity in today's computing systems, the autonomic computing model arises in response to the increasing difficulty of managing systems well beyond managing individual software environments [28]. An autonomic computing system can manage itself given high level directives, such as policies, from its administrators. These systems continuously monitor their operation in order to detect changes in internal or external conditions affecting the fulfillment of their quality attributes; then, autonomic systems adjust their operation to guarantee continuous alignment with high level objectives [28].

At the core of autonomic computing is self-management, as an approach to reduce human intervention in operative and detailed system administration and maintenance tasks. These tasks are classified in four properties of self-management: **self-configuring**, the ability to realize automated configuration of components and systems from high level policies; **self-optimizing**, the ability for components and systems to continuously search for optimal efficiency and performance improvement; **self-healing**, the ability of the system to discover, diagnose and recover from software and hardware faults; and **self-protecting**, the capacity of the system to anticipate and

defend itself from malicious attacks or unplanned cascading failures.

In [28] IBM researchers Kephart and Chess presented an architectural approach to self-managing systems, proposing the structure of an autonomic element; this approach contains the elements comprising the MAPE-K reference model, namely: Monitor, Analyzer, Planner, Executor, and Knowledge base. Each of these elements has specific responsibilities to accomplish system-level adaptations, either by structural or behavioral modifications of the system itself. Following are the functions of each element [8]:

Monitors sense relevant context information from the target system (*i.e.*, the managed system), such as service latency and throughput, or data about the current state of the computing infrastructure;

Analyzers analyze context data and determine whether or not an adaptation must be performed;

Planners synthesize a set of actions (*i.e.*, the adaptation plan) to alter the system's behavior, in accordance with the adaptation symptoms identified by the Analyzer;

Executors realize the adaptation plan through the available adaptation mechanisms, such as architectural reconfiguration and parameters tuning; and

Knowledge base is a set of data sources enabling data sharing —required to make self-management decisions— among Monitors, Analyzers, Planners, and Executors.

The relevance of self-managed software conception and vision in this thesis is that it constitutes the foundational strategy for our solution, even though we are focused on the monitoring aspect.

2.3 Domain Specific Languages (DSL)

As its name implies, a DSL is a specific purpose language whose syntax and semantics are tailored for specifying software artifacts (*i.e.*, *program specifications*) in a particular application domain (*e.g.*, testing, monitoring, security). It is designed to provide a specific notation to express solutions at different levels of abstraction using the vocabulary of the application domain. As a result, a well designed DSL is more flexible and effective than a traditional library, improving programmer's productivity, maintenance costs, and communication with domain experts [21]. Moreover, stakeholders can gain the understanding to validate and modify these program specifications [32].

DSLs increase not only productivity but also flexibility and reliability in software systems [29]. Developing a DSL can lead to automatic code generation and assembly, producing less error prone solutions. However, despite these advantages, Deursen and Klint [41] mention some maintenance disadvantages, including: first, using a DSL implies a shift from maintaining *hand-built*

applications to maintain (i) DSL programs specifying each application (although possibly with reusable elements), (ii) the DSL compiler, and (iii) the DSL library containing the basic set of util objects. Second, although programming languages are more difficult to learn and use, there are plenty of learning material such as courses, manuals, tutorials, and experienced professionals; in the case of a new DSL, all this material must be created by the DSL developers. Additionally, other authors consider as a disadvantage the unfamiliarity on how to fit DSLs into a regular development process [23, 39].

An important part of the contributions of this thesis is based on DSLs.

2.4 System Monitoring

In face of increasingly competitive markets and a continuous raise in the demand of pervasive services, companies are concerned with measuring and improving operational efficiency based on data gathered directly from the infrastructure itself. This has raised a demand for advanced mechanisms providing continuous monitoring of systems supporting their business activities [23]. Continuous monitoring differs from other techniques to measure performance (*e.g.*, profiling) in that its ultimate goal is not related to individual measurements, but to the permanent application of them; from the set of obtained measurements, a single value is calculated and compared with a reference value, such as a Service Level Indicator. Furthermore, detailed information about single measurements is useful to further analyze root causes of unexpected system behavior. However, capturing events can become difficult to handle in real time applications [31, 45].

The main objective in implementing system monitoring is to provide the means to report high-level activities that allow further processing to answer business-level questions, such as "what is causing the system to fail fulfilling the X quality agreement?" or "Why after the last system update user searches were reduced by Y%?". Answering these questions requires tools allowing to view the system in terms of how it is used, instead of how it is built, which is commonplace for monitoring technology [31].

Monitoring the System Behavior

In order to measure and control operational efficiency, tools and mechanisms are required to evaluate and assess the system state, behavior, and overall wellbeing and health. Currently, the main approach to monitor and assess such variables is through the use of metrics. Metrics are captured in critical measurement locations defined by the system's constituent components, allowing to characterize quality in service provisioning [35].

Effective monitoring mechanisms should consider at least two major tasks: the events

measurement process, and the events correlation and assessment process. The measurement process is performed by *in situ* sensors that generate events containing low level data related to service executions, such as execution times, memory consumption, and exceptional behavior. The correlation and assessment process gathers measurements from different sources (*i.e.*, sensors), composes them and computes single values. In case any unexpected behavior is found, monitor elements report the events to other MAPE-K elements. However, characterizing system behavior is not a trivial task, systems typically produce zillions of events per unit of time, which makes understanding system behavior a still open challenge [31]. This means that current technology only enables systems to see the events, not to make sense of them.

A first step towards understanding system behavior is tracking events causality, that is, being capable of tracing incidence of one event in the occurrence of subsequent events. Event causality is called *horizontal* when the causing and caused events happened at the same conceptual level in the system, and *vertical* when the causality relation is determined by the different layers composing the system [31]; in the latter, events are commonly referred to as low and high level events, depending on how close they are to the business layer. From causality tracking, complex patterns of events can be identified, and characterized as important behavior to report. Moreover, discovering these patterns and correlating them with contextual information, such as the current system state or the season of the year, can help avoiding certain behavior by generating appropriate counter-effects into the system.

2.4.1 Profiling, Monitoring and Tracing

Profiling, monitoring, and tracing are techniques used to identify specific behaviors in a system under operation. However, each of these techniques is valuable for different reasons, and its main purpose is specific to a certain concern. First, monitoring is usually an ongoing activity that is continuously observing elements or properties of the system. Based on that, when monitored elements fall below or above specific values, the monitoring mechanism triggers alerts, notifying interested stakeholders. Second, software profiling is a technique for dynamic program analysis that measures resources usage, such as CPU time, memory, and I/O, as well as execution time. This technique is usually for identifying code candidate for performance optimization. As it is a resource demanding activity, profiling is performed in development environments, as opposite to software monitoring which is (usually) used in production. However, program profiling can be also done using logs and/or monitoring data. Lastly, software tracing refers to trace the execution of a program. Program tracing is usually used in software testing, and it can be useful in different type of scenarios, such as call tracing, which helps in determing why a program is failing or not responding as expected, code coverage, which records which parts of a program were executed in a test suite, and live debbuging, which allows to execute a program instruction by instruction in

order to identify programming errors.

2.4.2 Monitoring System Performance

System performance depends upon the nature of the resources used to fulfill requests, and how shared resources are allocated when multiple requests occur in the same resources [6]. According to the performance taxonomy presented in [6], the performance concerns or requirements used to specify and assess the performance of a system are latency, throughput, capacity, and modes. These concerns are described below:

latency refers to a time window during which an event must be processed and a response must be produced.

throughput refers to the number of responses that have been completed in a given observation interval.

capacity is a measure of the amount of work that a system can perform. It is usually defined in terms of throughput. In that case, capacity refers to the maximum achievable throughput without violating latency requirements [9] (as cited in [6]).

modes refers to the response of the system performance in face of different (or changing) scenarios regarding latency, throughput, and capacity.

These performance concerns help in specifying the expected performance of a system from different fronts. Moreover, they can be tailored to specific needs, in a way that performance measurements have a particular meaning depending on the problem that the system solves.

2.5 Software Deployment

Software deployment is a post-production process that takes place between software acquisition and execution, and is performed for or by the software user [16]. Deploying a software system may be considered to be a process consisting of a set of interrelated activities within the deployment life cycle.

2.5.1 The Software Deployment Life Cycle

Hall *et al.* present in [24] a software deployment life cycle composed of several interrelated activities. Figure 2.1 depicts the different states of a system during its deployment life cycle, in which transitions are deployment activities. These activities are classified into two roles: software producer, and consumer. The first one consists in creating a packaged version of the software in order to provide a deliverable product, and the second one consists in configuring the provided package to execute it such that it can be used.

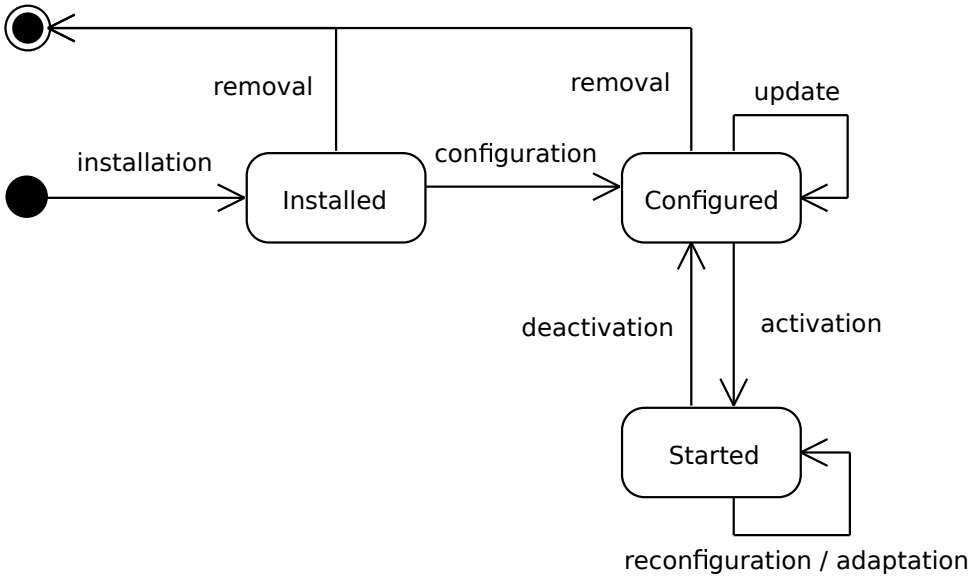


Figure 2.1: *Deployment Life Cycle. Adapted from [18]*

Producer-Side Activities

release is the bridge between development and deployment. This activity is in charge of all the necessary tasks to prepare, package, and provide a system for deployment in consumer sites. The released package contains the software resources, such as libraries, configuration files, and executables, as well as descriptors specifying the resources required to deploy the software.

retire is the process of removing support for a software system or a given configuration of a software system. Retiring a system makes it unavailable for future deployments.

Consumer-Side Activities

install is the initial consumer activity in charge of the configuration and assembly of all the resources necessary to use a given software system.

activate and deactivate are activities that allow the user to actually use a given software. For simple tools, these activities are usually realized through the creation of certain command, or clickable icon, for executing and stopping a binary component of the tool. Complex software may be composed of several components that must be executed in order to use it.

reconfigure, update, and adapt are the activities responsible for changing and maintaining the configuration of the deployed system. These activities may occur any number of times, in any order. The objective of the update activity is to deploy a new, previously unavailable software configuration. The reconfigure activity also changes a previous installed software, but selecting a different configuration. The adapt activity is in charge of monitoring the deployed

software system, and respond to changes in order to maintain consistency in the deployment software system.

remove is the activity performed when the deployed software system is no longer required at the consumer site. This activity implies the undo of all changes caused by previous deployment activities at the consumer site.

2.5.2 Build Automation Software

Before a software product can be used by its intended users, the software developers must compile the source code for a particular target platform; that is, build a software release. The process of compiling source code takes as input individual source code files, or a parent directory containing all the files composing a program, and generates one or several binary artifacts that are then executed or interpreted by another program. When the source code makes use of third-party libraries, the compiler must know these libraries as well in order to find errors in the program and optimize the generated binary artifacts.

As the software grows, the compilation process can become difficult, time consuming and error prone. Furthermore, as developers may work on different subsystems, manually compiling the whole software system will eventually worsen the scenario. To overcome this, building the software system is generally automated using scripts or more advanced tools such as Make, Maven, or Ant. Build automation is different than deployment automation, continuous integration, and continuous delivery. These processes are focused on deploying or installing a release into a particular environment, build a software product as developers check-in changes in the source code, and a combination of both, respectively.

Chapter 3

Problem Domain Analysis: Dynamic Performance Monitoring

In this chapter we analyze the high-level functional requirements for designing and implementing our envisioned dynamic monitoring infrastructure. We start by specifying the requirements for the Monitor and Sensor elements of the MAPE-K reference model [11]; then, we continue with the definition of the requirements for our dynamic monitoring infrastructure, inspired by DYNAMICO's monitoring feedback loop [43]. finally, we identify a set of quality considerations that are relevant for the design of our monitoring infrastructure.

3.1 Requirements Elicitation

3.1.1 MAPE-K: Monitoring Requirements

Our aim at developing this thesis is to provide standard mechanisms to monitor software applications' performance in dynamic contexts. Aligned with the vision proposed in the DYNAMICO reference model [43], our approach needs to support changing values and thresholds of the context variables and the monitoring logic that capture them (*cf.* Section 1.3). As the feedback loops present in such reference model are based on the MAPE-K reference model [11], a first approximation to the design and implementation of DYNAMICO's dynamic monitoring infrastructure is to consider the Sensor and Monitor elements from the MAPE-K reference model. Nonetheless, we identified a lack of a detailed and standard reference specification and architectural design for such model. In light of this, in [3, 4] we presented a base design for building autonomic systems using MAPE-K, including the structural and behavioral architectural designs.

IBM's architectural blueprint for autonomic computing describes the high-level functions composing the internal structure of an autonomic manager [11]. From IBM's proposal, we have identified a set of functional requirements for the elements composing the MAPE-K reference model. Although

the requirements in this section are focused on the monitoring part only, the rest can be found in [4]. The following requirements are split in Sensors and Monitors. On the one hand, sensors are elements consisting of a set of properties exposing the current state of certain manageable resource, and a set of events that occur when the manageable resource's state changes. From now on, we refer to sensors as probes, since that name is more accurate to our approach. On the other hand, monitors are elements that collect details from the managed resources, using a standard manageability interface to probes, and correlate them into symptoms that can be analyzed [11].

Software Probes (also called Sensors) Requirements

S1 A probe must collect measurements of variables of interest (from now on, referred to as sensed data) (e.g., quality attributes specified in the series of standards ISO 25000 such as performance of a service, availability of resources, topology information, configuration properties) [1] in the context in which it is located, that is, its execution context or the context of the domain to which it belongs.

S2 A probe must temporarily store sensed data.
Rationale. The monitors' responsiveness relies on the timely availability of the sensed data. This availability can be achieved by supporting temporary storage, which would allow monitors to gather data at any moment. Nonetheless, a drawback of this approach is that probes can use memory space that should be available to the target system, thus, other storage options should be considered.

S3 A probe must expose a subset of the sensed data to the set of monitors, in both cases: when monitors and probes have been deployed jointly and when they have been deployed independently.

S4 A probe must remove a subset of the sensed data being stored temporarily when instructed by a monitor.

S5 A probe must perform primitive operations (*e.g.*, count repetitions of a measurement in a given time interval) on a subset of the sensed data.
Rationale. The ongoing transmission of sensed data from probes to monitors can overuse network resources, thereby hindering the target system's regular operation. Placing primitive operations in probes can considerably reduce the amount of data transmitted through the network when monitors do not require the entire collection of sensed data but, instead, computations on it.

Monitor requirements

M1 A monitor must obtain the sensed data from one or more probes, where it has been captured, through the required access modes, that is, by request (pull) or per occurrence (push).

M2 A monitor must compute metrics (based on sensed data) related to the variables of interest to characterize the current state of the target system. These calculations can be triggered periodically or per measurement occurrence, can produce average or instant calculations, and this calculation can also involve the composition or correlation of metrics calculated by other monitors.

M3 A monitor must make the calculated metrics available, through a *Knowledge Manager* element, to other monitors so they can compose their own calculations.

M4 A monitor must filter the calculated metrics before being reported to the Analyzer element. The filter must be applied through a set of domain-dependent monitoring rules over the calculated metrics.

M5 A monitor must report to the Analyzer element control symptoms, i.e. the metrics (simple or compound) that meet the conditions set by the monitoring rules.

M6 A monitor must allow changing the periodicity in which it calculates its metrics.

3.1.2 Dynamic Monitoring Infrastructure Requirements

In the previous subsection we specified the functional requirements for the monitor and probe elements according to IBM's blueprint for autonomic computing. Varying monitoring requirements is not a concern in such proposal, therefore, the requirements above are insufficient in our effort for implementing the monitoring infrastructure to support dynamic capabilities specified for the monitoring feedback loop in the DYNAMICO reference model. Subsections 3.1.2.1 and 3.1.2.2 present the requirements of the Dynamic Monitoring Infrastructure, along with corresponding quality considerations.

3.1.2.1 Functional Scope

We extend the functional requirements of the monitor element in order to orchestrate a dynamic monitoring infrastructure, composed of four stages: data acquisition, data aggregation and filtering, data persistence, and data visualization. Our approach to the first two stages is to provide tailored syntax and semantics in order to separate application code from monitoring logic, and to ease the specification of probes and monitors. Regarding the last two stages, as context variables may differ in their nature, it is necessary to consider different data storage and visualization technologies. Because of this, these stages are considered as pluggable elements of infrastructure.

Data Acquisition

M7 The definition of a monitor must specify the probes it requires to acquire measurements regarding its variables of interest.

M8 Probes must be deployed independently from the target system's components. This implies that already deployed components can start being monitored at any time.

M9 Probes must be capable to intercept different types of events associated with service execution, including: invocation, return, execution time, network communication time, and/or exception.

M10 For custom events raised at a lower level than service execution there must exist a library to program custom probes. These custom probes must be accessible in the same fashion that built-in probes.

M11 Developers must specify how to create, update, and retrieve measurement values for custom metrics. In the case of retrieval operations, define whether access mode is by request or per occurrence.

Data Aggregation and Filtering

Application developers are concerned with the design and development of the mechanisms for aggregating and filtering the monitoring data of interest; and at the same time, they are concerned with the integration of existing code bases in order to increase productivity and quality. However, crosscutting code compromises the comprehension and evolution of the monitoring components [23]. Accordingly, the monitoring infrastructure should provide standard mechanisms with established protocols to:

M12 Localize, use (*i.e.*, get and set values), and share context variables through a standard mechanism. Such mechanism must support the addition and removal of variables at runtime.

M13 Manipulate collections of events and perform calculations over them.

M14 Define reference values (*e.g.*, service level indicators) and compare measurement values against them in order to notify external services about unexpected behaviors.

M15 Attach new measurements with contextual information.
Rationale. Tagging measurement data would allow to categorize variables' values, giving meaningful information to search and filter measurements for visualization purposes.

M16 Use existing class libraries in order to perform calculations or invoke existing APIs in order to aggregate and/or filter measurements.

M17 Specify handling logic for service execution events, periodic events (*i.e.*, time-based events), and changes in context variables.

M18 Update the set of monitoring rules it applies to perform the filter of metrics.

Data Persistence

M19 The monitoring infrastructure must support persisting context variables on disk, along with their contextual information. The persistence mechanism should be applied independently from the monitors updating the context variables.

M20 Different persistence technologies can be applied to different variables.
Rationale. The widespread use of NoSQL technologies has promoted the appearance of simplified databases aiming at improving performance and scalability in data storage and querying; third-party services and libraries for managing logging and metrics information, for instance, are a relevant option.

Data Visualization

M21 Developers must be able to create visualizations (*e.g.*, charts) summarizing historical measurements associated with the context variables.

M22 Data visualizations must allow to graphically represent data associated with the built-in events raised in probes.

M23 Developers must be able to pre-process (*e.g.*, filter, transform, correlate) historical measurements in order to provide visualizations of relevant data.

3.1.2.2 Quality considerations

Relevant quality considerations include the compatibility, co-existence and interoperability implied by dynamic deployment and re-deployment of monitors and probes, scalability of the infrastructure, and standard composability and controllability mechanisms. By composability we refer to the quality property characterizing software artifacts that can be composed and reused effectively, in order to maximize their cost-effectiveness [26]. Tables 3.1, 3.2, 3.3, and 3.4 detail the quality scenarios associated with these considerations. These quality scenarios are based on the definitions of quality attributes in [6].

Quality Scenario	*1.* Interoperability in dynamic deployment and re-deployment of monitors and probes
Quality attribute	Compatibility, Co-existence and Interoperability
Justification	Deployment and re-deployment tasks can lead the Monitoring Infrastructure into an erroneous state given that components (*i.e.*, probes and monitors) can be re-inserted into the same running middleware, or context variables can be defined more than once. Moreover, updates to the target system's components can stop probes from continue working.
Stimulus	An already deployed Monitor is re-deployed.
Source of stimulus	A change in the monitoring logic and/or in the context variables.
Artifact	Monitoring infrastructure components responsible for generating and deploying the new components.
Response	The Monitoring Infrastructure performs the necessary actions to support the (possibly) generation of new probes and monitors, and ensure that they are deployed correctly. If they were already deployed but require logic modifications, to undeploy them before re-deployment, and ensure that variables are defined only once.

Table 3.1: *Quality scenario for Co-existence and Interoperability of Monitors and Probes*

Quality Scenario	*2.* Scalability of the Monitoring Infrastructure
Quality attribute	Scalability
Justification	Continuous monitoring can generate large amounts of events and logging information, which requires mechanisms to support taking advantage of new computational resources, if available. Components that are being monitored can reach an unresponsive state.
Stimulus	Monitors and probes reach critical usage levels of computing resources' capacity (*e.g.*, disk, memory, network).
Source of stimulus	target system's services are on high demand.
Artifact	Monitoring infrastructure components responsible for accounting and deploying the new or existing components in new computational resources.
Response	Some components of the Monitoring Infrastructure (*e.g.*, Monitors) are deployed in new computing resources.

Table 3.2: *Quality scenario for Scalability of the Monitoring Infrastructure*

Quality Scenario	*3.* Composability of monitors
Quality attribute	Modifiability - Modularity
Justification	Monitors must be capable of effectively reusing already existing monitoring elements, such as probes, in order to increase development productivity.
Stimulus	A monitor needs to receive measurements from an already existing and deployed probe.
Source of stimulus	A new monitor is under development as response to emerging monitoring requirements.
Artifact	The architecture of the dynamic monitoring infrastructure supports the composability of generated monitor components.
Response	Probes are identifiable and targetable within the monitoring infrastructure, in a way that new monitors can subscribe to them for receiving new measurements.

Table 3.3: *Quality scenario for Composability of Monitors*

Quality Scenario	*4.* Controllability of Monitors
Quality attribute	Modifiability - Changeability
Justification	Changes in the Monitoring Infrastructure include not only modifying the measurement strategies or monitoring logic, which would require re-deployment, but also (re)configuring the execution at runtime.
Stimulus	A re-negotiation of the Service Level Indicators causes a modification on the calculation period of certain measurement.
Source of stimulus	The target system's context
Artifact	A monitor performing periodical calculations on the sensed data.
Response	Monitors expose a manageability interface to allow external components modifying the calculation period for a given measurement.

Table 3.4: *Quality scenario for Controllability of Monitors*

Chapter 4

Solution Domain: A DSL Approach to Dynamic Monitoring and Deployment

In this chapter, we introduce the global design of our solution. Then, we explain the implications of the related quality concerns on the proposed design, and provide an overview of the resulting monitoring infrastructure. Finally, we explain in great detail the design of Pascani and Amelia, two domain-specific languages composing our solution.

4.1 Overview of the Global Architectural Design

In Chapter 3, we introduced the functional requirements to realize our envisioned Dynamic Monitoring Infrastructure, and related quality concerns. In this set of requirements, we identified two important groups: the first, related to the software monitoring tasks, and the second, to the deployment of these monitoring artifacts. We address these requirements and quality concerns by designing two domain-specific languages named Pascani and Amelia, respectively.

Figure 4.1 illustrates the component-based model of our global architectural design. From a set of Pascani monitor specifications, the language engine generates SCA-compliant monitoring components implemented in Java, and the corresponding Amelia deployment specification. From a deployment specification, the Amelia language engine generates an executable Java program able to communicate with UNIX-based computing nodes in order to execute the necessary operations to deploy the specified SCA artifacts. Once the monitoring components are running in the infrastructure, along with a set of provided monitor probes, the infrastructure continously gather performance data from the Target System. This is done by sending measurements from probes to monitors, using either pull or push communication. Either way, this data is then used to update the context variables, which in turn are stored in one or several databases. From there, the IT personnel can visualize the current system performance, modeled as context variables, using different dashboard technologies. To better understand our solution approach, we further explain it using

an example in the next section, which gives more detail on how the proposed architecture supports the development of monitoring strategies for an online retailing system.

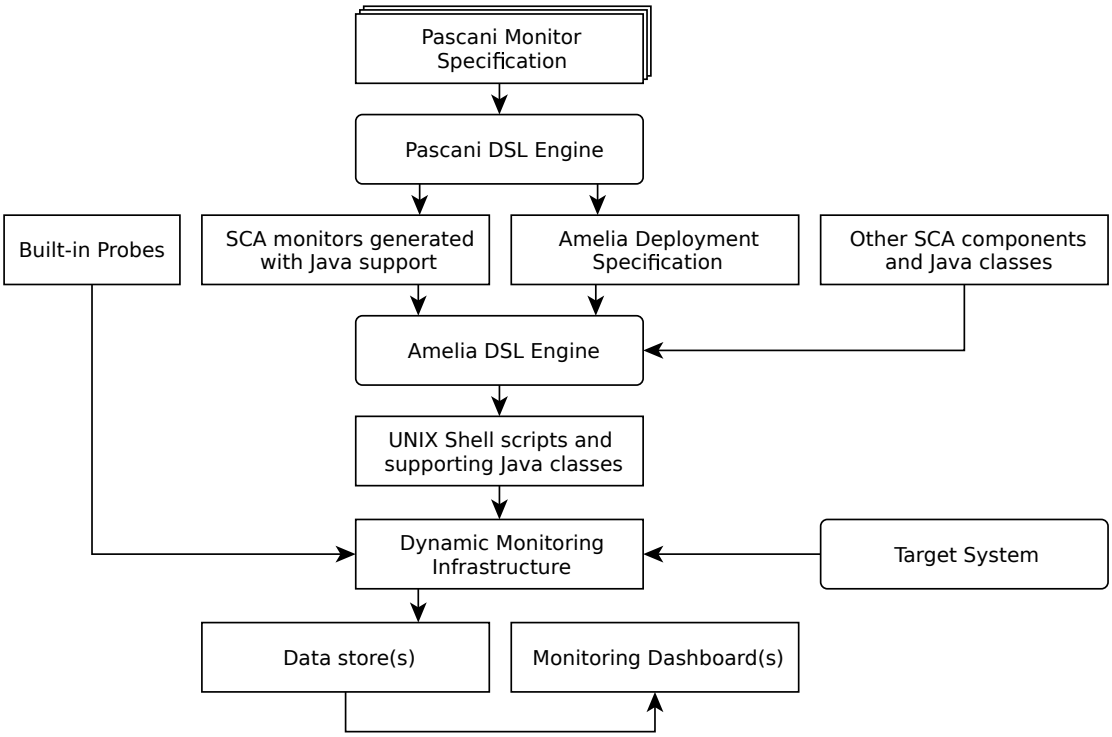


Figure 4.1: *Global architectural design (informal) of our proposal*

Dynamic Monitoring: The Online Store Retailer Example

In e-commerce, an Online Store Retailer (OSR) application allows customers to shop for products and services through the Internet. Users browse a catalog of products and select the ones they want to purchase by adding them to a shopping cart. Furthermore, with the proliferation of personal context and user-centric recommendations, users expect from OSR applications product recommendations based on their preferences. To complete the purchase, customers initiate the checkout process by entering information about delivery and payment preferences. Delivery preferences include information such as the shipping address and the preferred delivery service. Payment information includes data about the preferred payment method such as billing address, the credit card number, security code, and holder name. OSR applications typically compose several third-party services to recommend relevant products, complete the checkout process, and confirm the order delivery. In a similar way, for completing the checkout process, OSR applications use a postal address verifier service to verify that the shipping and billing addresses do exist; a credit card verifier service to confirm all the information about credit cards; and a delivery service, with its corresponding delivery tracking service, to dispatch the order and for the user to monitor

the delivery process. Different software vendors already provide instances of these services to be used manually by humans, and programmatically through public APIs.

Assume that a local business has been receiving complaints from their customers during the last two weeks. According to their ticketing system, there are sporadic long delays when finalizing a purchase in their OSR application. After deeper conversations with the affected customers, and some infrastructure analysis, the IT personnel assures that this issue must be related to one of the third-party services involved in the checkout process. In fact, the delays are being caused by the credit card verifier service, but their current monitoring solution does not provide any insight about the real problem, as it is limited to provide information about their computing infrastructure. Since the long delays cannot be predicted or reproduced, the initial tests they executed on the third-party services are not helpful in demonstraiting that the service provider is not fulfilling the agreed quality. To discover what is causing the performance issues, we use PASCANI as follows.

1. Application developers identify the context variables that allow modelling the performance issue in the checkout process. In this case, measuring service latency is certainly enough for detecting which of the services is causing the long delays in the checkout process.

2. Having into account the Target System's components (*i.e.*, the high-level elements composing the OSR application), the application developers create a PASCANI performance monitor with logic to introduce a probe into each target service at runtime, and to gather the necessary information to update the latency context variable.

3. Once the monitor specifications are finished, the application developers execute the PASCANI engine to generate the SCA monitor and its corresponding AMELIA deployment specification. Then, they also execute the AMELIA engine for generating a Java program to deploy the generated monitor.

4. In order to start monitoring the OSR application, the application developers execute the Java program generated by the AMELIA engine; this program compiles the generated source code, transports the resulting artifacts to the computing infrastructure, and then executes the monitoring components.

5. Since generated SCA monitors are introduced into the running infrastructure without stopping the Target System, the monitoring logic is put in place immediatly. Every time a service is executed (*i.e.*, a custumer is finishing a purchase), the context variables are updated and their new values are populated to the data stores.

6. While the monitoring infrastructure is gathering data from the Target System, and populating the context variables' values to the data stores, application developers create helpful visualiza-

tions such as stacked bars or gantt charts to discover what service is causing the performance issue.

By visualizing the latency associated with each target service, the application developers can now discover the source of the unexpected delays. After having discovered which service is not performing according to the service level agreements, the personnel from the local business can make decisions about the service provider, such as triggering default clauses in agreements of service quality. In a similar way, the service provider can analyze the issue, and discover if this is a problem in their infrastructure, or it comes directly from the credit card company's API.

4.2 Addressing Quality Concerns

This section explains how we address the quality concerns presented in Subsection 3.1.2.2.

Compatibility, Co-existence and Interoperability of Monitors and Probes

Table 3.1 details the effects of introducing new and already existing components into the running infrastructure. This scenario can be broken down into two cases: the introduction of new monitors and probes, and the deployment and re-deployment of monitors and probes. Regarding the first case, introducing new elements into the Target System would generally entail a compatibility check at design time; however, our envisioned dynamic monitoring infrastructure requires the introduction of new elements at runtime. That is, we need to augment the service execution processing with measurement code whilst service requests are happening. This is a proper scenario for using the Interceptor design pattern, as long as the Target System's middleware supports it (*i.e.*, adding interceptor elements at runtime). Examples of this are Intent components in SCA, and the use of Aspect Oriented Programming in EJBs, through Java annotations or XML configuration files, and OSGi, through the AspectJ extensions. Figure 4.2 shows the use of this design pattern to produce measurement data, and to provide access to monitors through the `Probe` component. Regarding the second case, it would require not only adding new interceptor components at runtime, but also removing them. As communication between monitors and probes is bidirectional, this can be source of errors because of broken bindings at runtime; thus, this places a new design concern regarding static dependencies bindings. Static bindings require service provider and consumer to be bound at system deployment, and cannot be replaced at runtime. In this case, monitors and probes need to be decoupled, in a way that *i)* measurement data is exposed without knowing the components consuming it, and *ii)* measurements are published without knowing the components making use of them. These use cases match with the context of application of the Publish/Subscribe design pattern.

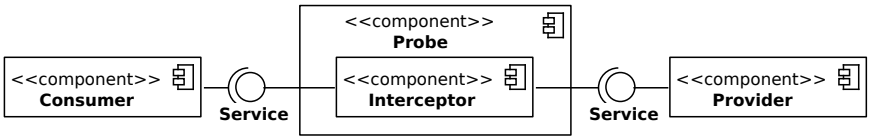


Figure 4.2: *Interceptor Design Pattern applied to Probes components*

Scalability of the Monitoring Infrastructure

Achieving scalability in a Dynamic Monitoring Infrastructure requires the ability to deploy monitors in new computational resources, independently from the Target System's components. It also requires to handle the always increasing amount of measurement data, when supporting in-memory storage, or when the Target System is under heavy load. This can be done by introducing the Message Broker architectural style, which would decouple message senders and receivers (*i.e.*, probes and monitors), as discussed above, and would allow distributing the Monitoring Infrastructure's components across multiple computing resources. Furthermore, automatically moving components across different computing resources would require the design and implementation of an autonomic infrastructure, with special relevance in the planning function. The scope of this thesis, nonetheless, is limited to the automation of the deployment mechanisms needed to realize the resulting reconfiguration plans of such infrastructure.

Composability of Monitors

Composable monitors refer to self-contained (modular) elements that can be reused effectively, with the aim of increasing software quality, reducing maintenance costs, and improving productivity of the development team. Such result is not easily achievable, the monitoring logic depends upon many factors, including variables of interest, the components from which those variables are calculated, the Target system's architecture, among others. These dependencies make difficult to create configurable components that can be selected and assembled to build new monitor components. Although this makes difficult to reuse monitors as a whole, we can provide them with desirable attributes to promote composability on different fronts [34]:

- Sound architecture: good interface design (particularly for visible interfaces) and architecture structure can considerably facilitate composition,
- Abstraction: providing an appropriate level of abstraction in the specification of the monitoring logic can simplify the abstraction of related technical layers. This can be very beneficial for composition of monitoring components, when there is suitable independence of the provided monitoring concepts,
- Modularity: a monitoring infrastructure based on specific abstractions encapsulated into well-defined entities promotes more modular monitor components. Modularity is very beneficial

for composability if interfaces and specifications are well defined.

Controllability of Monitors

Controlling the execution of the Monitoring Infrastructure allows timely reacting to changes in the Target System's context. Critic levels of memory usage in the Target System, for instance, would require to reduce the memory footprint of monitors and probes deployed in the same computing resources. Other critic scenarios require modifying the periodicity of measurement metrics calculation, and even temporarily stopping all monitoring activity. These controllability requirements are addressed by providing all elements of the Dynamic Monitoring Infrastructure with manageability interfaces or methods to remove either portions or all of the sensed data (probes), and modify the periodicity of metrics calculation (monitors). We also provide these and other elements of the infrastructure with an interface to pause and resume all monitoring activity, including propagation of measurement data and changes in the context variables.

Traceability and Logging of Monitors

Reporting fine-grained levels of information about how is the infrastructure performing is essential to discover sources of errors and undesired behaviors. Moreover, traces of the execution and detailed data on transactions is useful to find anomalies' causes. In the monitoring infrastructure, we have designed an event-based logging system that propagates log and deployment events raised at each component location. These events are also stored, allowing to reproduce the series of events in a given time window. This also allows visualizing logs as a timeline of events, with the possibility of filtering by logging levels.

4.3 Design Overview of the Dynamic Monitoring Infrastructure

Section 4.1 presents the global architectural design of our solution, composed not only of the monitoring elements, but also of its corresponding deployment components. In this section, we give a design overview of the monitoring infrastructure and its composing elements. This design incorporates the design decisions we analyzed in Section 4.2 with regard to the infrastructure's quality considerations.

The most relevant elements composing our monitoring architecture are Monitors, Probes, and Namespaces. The first two elements follow the considerations we have discussed in previous chapters. Namespaces are in-memory stores for values associated with names; these elements allow registering context variables at runtime, as well as getting and updating their values. Figure 4.3 depicts the basic behavior for each of these elements, and the manageability interfaces to control their execution, as discussed in Section 4.2. These three elements are conceived as SCA components, which not only standardizes but also simplifies the build, deployment, and management of our

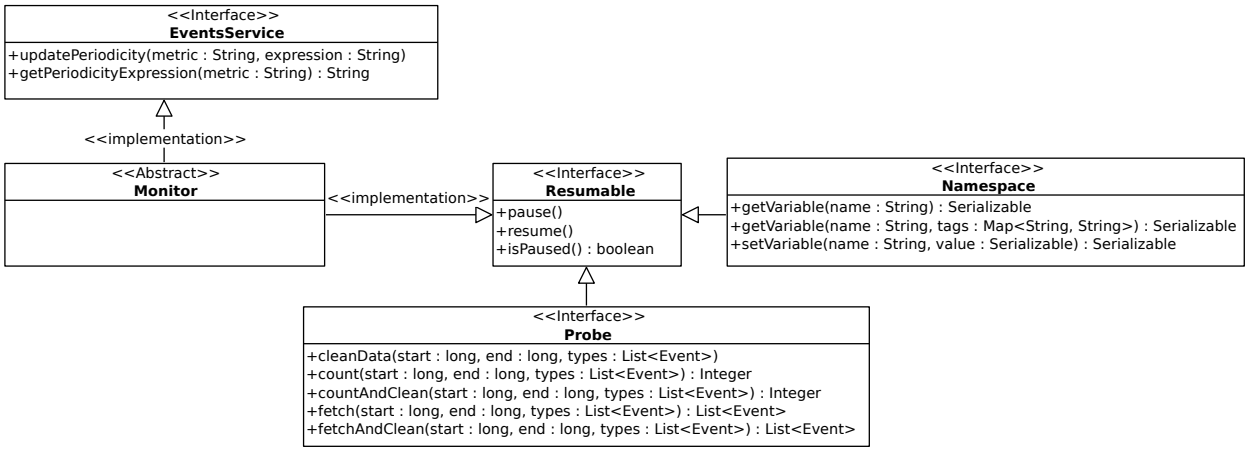


Figure 4.3: *Main elements composing the Monitoring Architecture*

infrastructure.

The `Probe` interface in Figure 4.3 exposes a set of basic operations to fetch, count, and clean measurements performed within a given time window. This allows monitors to pull measurements (*i.e.*, events) from probes, and perform calculations to update the context variables. As analyzed in the subsection *Compatibility, Co-existence and Interoperability of Monitors and Probes*, monitors and probes need to be decoupled using the Publish/Subscribe design pattern (cf. Figure 4.4). In order to mediate between publishers and subscribers, we use a distributed message broker. In this communication scheme, monitors subscribe to measurements of interest, and possibly to changes in context variables. This data is published by probes and namespaces, respectively. Since we decided to remove static bindings, pull communication is done through remote procedure calls (RPC) using also the Publish/Subscribe design pattern.

Following the same communication scheme, we add persistence and log-ability to the monitoring infrastructure. Each element within the infrastructure produces in-place logs that are published through the message broker. Persistence is realized by the `Data Store Mapper` component, which is subscribed to all changes in context variables, new log information, and new deployments of monitors and namespaces.

Sections 4.4 and 4.5 present the design of Pascani and Amelia, the two domain-specific languages we introduced in the *Global Architectural Design* section.

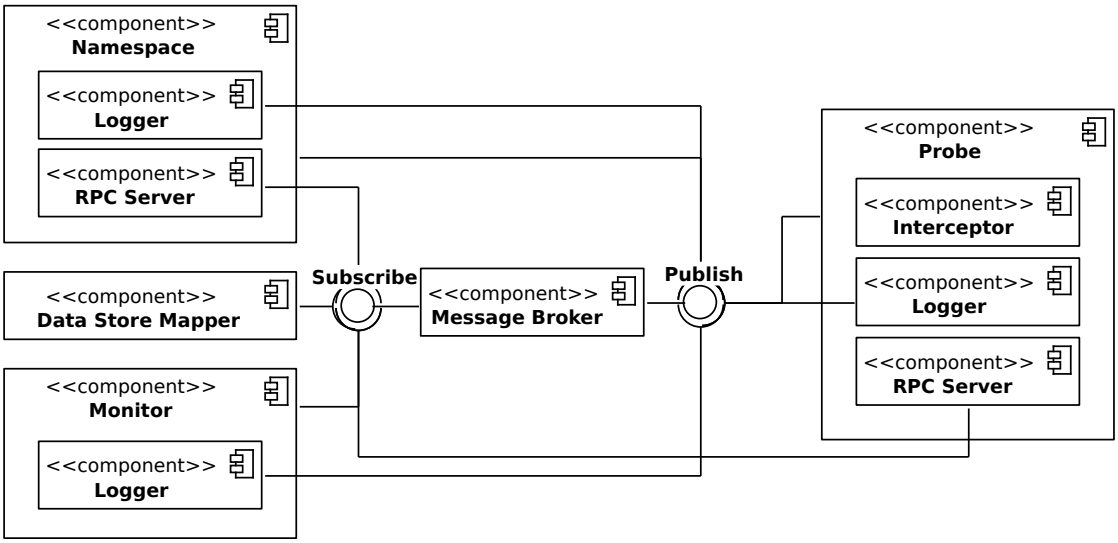


Figure 4.4: *The Publish/Subscribe design pattern applied to elements of the Monitoring Infrastructure*

4.4 PASCANI: A DSL for Dynamic Performance Monitoring

PASCANI is a component-based and statically-typed Domain Specific Language for specifying dynamic performance monitors for component-based software systems. It is tightly integrated with the Java type system, which allows the integration of already existing libraries into the language. It also provides some of the Java control statements with a more flexible syntax, based on the Xbase expression language [19].

From PASCANI specifications, the language implementation generates the artifacts for the Dynamic Monitoring Infrastructure, including their deployment specifications, as explained in section 4.1. The generated artifacts are composed of elements from the PASCANI runtime library and the SCA library, which are presented in sections 5.1.1 and 5.1.2. To complete the automation of the dynamic performance monitoring, the deployment specifications are executed by our second DSL, AMELIA, presented in Section 4.5

4.4.1 Language Concepts

The PASCANI language comprises two main concepts: monitors and namespaces. These concepts represent a textual abstraction to the Namespace and Monitor interfaces introduced in Figure 4.3. The semantics associated with each of these concepts are based on the behavior defined by such interfaces. Namespaces, as their name imply, are stores for values associated with names, identified with a store name. Each name inside a namespace corresponds to a context variable. Monitors are containers of events and event handlers, that specify the required monitoring logic to calculate metrics. These two concepts have been designed to be used together through metrics, which are

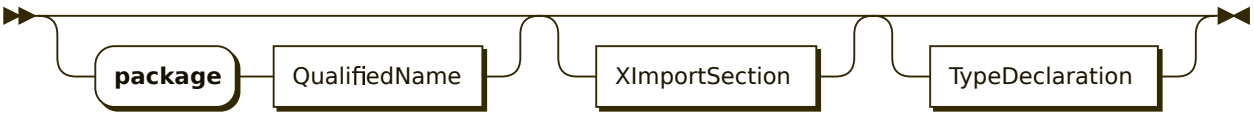
used to update the context variables defined in namespaces.

4.4.2 The Pascani Grammar Definition

The following syntax diagrams comprise a simplified version of the Pascani grammar definition. Syntax diagrams, or railroad diagrams, are a visual way to represent context-free grammars. Each diagram defines a non-terminal. A diagram describes possible paths between an entry point and an end point, by going through other non-terminals and terminals. Terminals are represented with round boxes, while non-terminals are represented with square boxes. The complete grammar is presented in Appendix C. Grammar rules starting with an X are inherited from Xbase unless they are explicitly re-defined within the grammar.

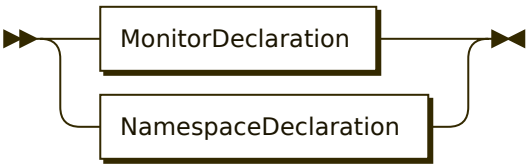
A valid file in Pascani has an optional definition of a package name, a Java import section and a type (of compilation unit) declaration section. This means that empty files are valid specifications that do not generate any artifact. A compilation unit is a translation unit, that is, a source file containing the definition of either a namespace or a monitor. These are the two compilation units in Pascani, that correspond to the Namespace and Monitor elements presented in figure 4.3.

MonitorSpecificationModel:



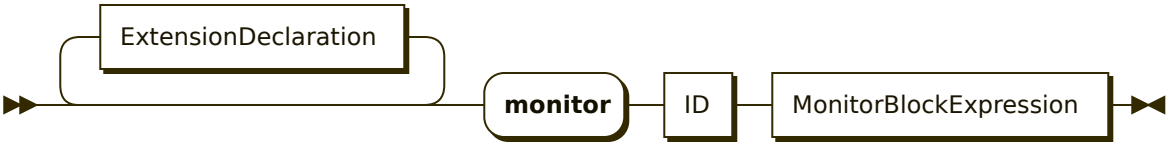
Each compilation unit has different syntax and semantics, and has been designed to be used with each other; using namespaces inside monitors makes transparent getting and updating context variables' values, which eases the development by letting developers focus on the variables instead of technical details. In the online retailing system example presented in subsection 4.1, a namespace would contain context variables such as service latency or throughput. Pascani releases application developers from maintaining and sharing the state of such variables. Moreover, as the infrastructure may be distributed among several computing nodes, it also makes transparent the communication protocols and technical details to get and update variables' values.

TypeDeclaration (Compilation Units):



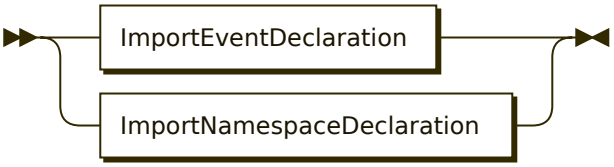
From a general perspective, monitors are made of an extension section, a name and a block of expressions. The monitor's qualified name (*i.e.*, the concatenation of package and name separated with a dot) must be unique in the project's classpath.

MonitorDeclaration:



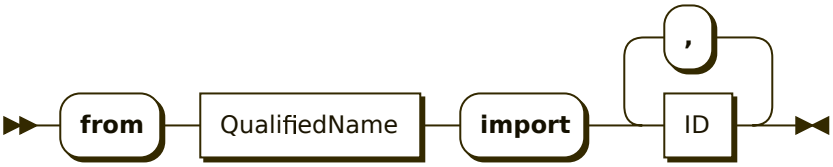
An extension declaration defines a point where the monitor is extended beyond its own expressions, being augmented with declarations from other types. An extension declaration is either an event import, or a namespace import.

ExtensionDeclaration:



One of the expressions inside monitors is the event declaration. When event declarations refer to execution events, they can be reused inside other monitors to avoid the unnecessary introduction of monitor probes into the Target System. This is done by explicitly importing events and treating them as part of the monitor. The event import statement uses the qualified name of the declaring monitor, and the name(s) of the event(s) that are being imported.

ImportEventDeclaration:



Importing namespaces allows monitors to read and update variables from different contexts. For instance, if a developer wants to model Service Level Indicators in PASCANI, she can create a namespace with the reference values, and another with the actual system values. This promotes separation and grouping of variables according to the different monitoring concerns.

ImportNamespaceDeclaration:

A namespace declaration contains the reserved word `namespace`, followed by a unique name and a block of expressions.

NamespaceDeclaration:

Listings 4.1 and 4.2 show the contents of two valid files written in PASCANI, following the grammar rules presented above.

```
1 package com.company
2
3 /*
4  * @date 2016/06/22
5  */
6 namespace SLI {
7    // Context variables
8 }
```

Listing 4.1: *Minimal example of a namespace specification*

```
1 package com.company
2
```

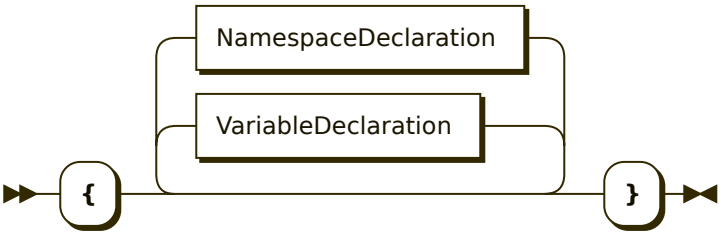
```
3 import java.util.List
4 using com.company.SLI
5
6 /*
7  * @date 2016/06/22
8  */
9 monitor Throughput {
10    // Monitor expressions
11 }
```

Listing 4.2: *Minimal example of a monitor specification*

Sharing of Context Variables

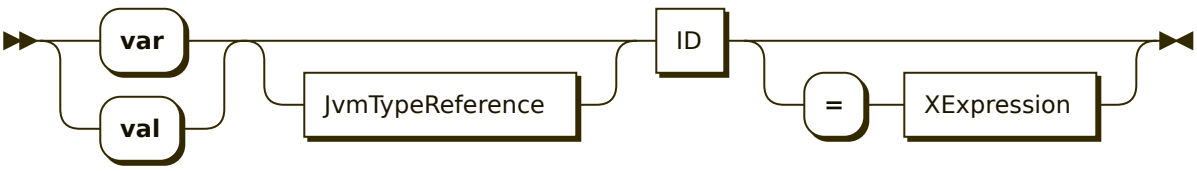
In Pascani, Namespaces are hierarchical name stores holding context variables. A namespace is composed of variable declarations and (optional) inner namespaces, which create the hierarchical structure.

NamespaceBlockDeclaration:



A context variable can be defined as an immutable value or a variable, and specifying its type is optional. In case no type is specified in the declaration, an initial value must be assigned to the variable. The right part of the declaration is an expression, which means that not only primitive types are allowed, but also any type by means of any valid expression in the language.

VariableDeclaration:



Variables support any type from the Java type system that is serializable, including custom types, as namespaces allow importing Java classes. Types must be serializable because the variables' values are sent across the network to different components, such as monitors listening for

changes in variable's values.

In a monitor, a variable is accessed using its qualified name. In this case, the qualified name of a variable is the concatenation of the hierarchical structure, separated with a dot between every namespace, finishing with the variable name. For instance, the variable *reference* in listing 4.3 would be accessed with `SLI.Performance.Throughput.reference`. It is worth noting that *reference* is an immutable value, therefore it cannot be modified but only read.

```
namespace SLI {
   namespace Performance {
      namespace Troughput {
         val reference = 100 // transactions per minute
         var Integer actual
      }
   }
}
```

Listing 4.3: *Example of namespace declaration with inner namespaces and variable declarations*

A variable's qualified name is used as the variable's name, and getting and updating its value is done as with a variable in any general purpose language such as Java. PASCANI also allows to attach contextual information to a variable's value; this is done by *tagging* the assignment using the class `TaggedValue` directly, or by using the helper method `tag`. Listing 4.4 shows an example on how to get and update a variable's value, including tagged values.

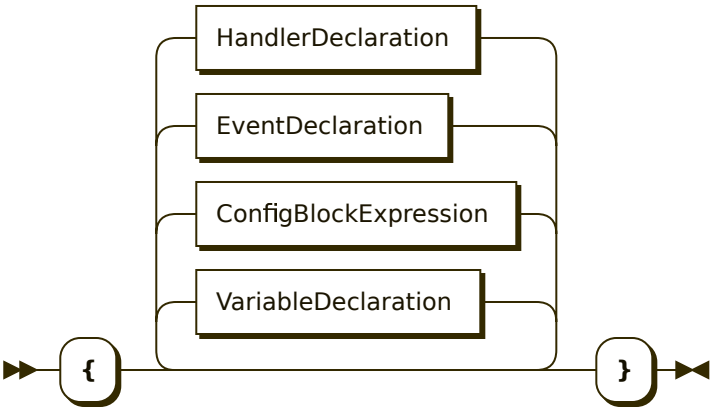
```
// Contextual information
val Server0 = #{ "node" -> "grid0", "component" -> "Server" }
val Server1 = #{ "node" -> "grid1", "component" -> "Server" }

// Update the value
SLI.Performance.Throughput.actual = 88
// Update the value and attach contextual information
SLI.Performance.Throughput.actual = tag(92, Server0)
SLI.Performance.Throughput.actual = tag(85, Server1)

// Get the current value
println("Reference value: " + SLI.Performance.Throughput.reference)
val Server0T = SLI.Performance.Throughput.actual(Server0)
val Server1T = SLI.Performance.Throughput.actual(Server1)
```

Listing 4.4: *Getting and updating variables*

Specification of Monitor Components

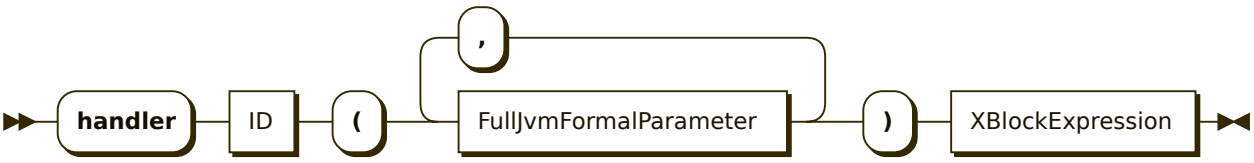
Monitor components are reactive by default. This is because all monitoring logic can only be triggered by an event (*i.e.*, there is no *main* method or entry point for a monitor to be invoked directly), whether it is a periodical event, an execution event, or a variable change event. A monitor specification can contain variable and event declarations, event handlers and configuration blocks. The way events and handlers work together follows the Implicit Invocation design pattern [22]; this means that events do not know its subscribed handlers neither their logic, and handlers can be added and removed at any time.

MonitorBlockExpression:



Event handlers receive two parameters, the first one is an event object and the second is a data map object. Each type of event has an event class with information about the event being notified. The first parameter allows accessing the event data from the handler's logic. The second parameter is optional, and contains information provided at the event listener subscription. It is useful when a single event handler is handling several events.

HandlerDeclaration:



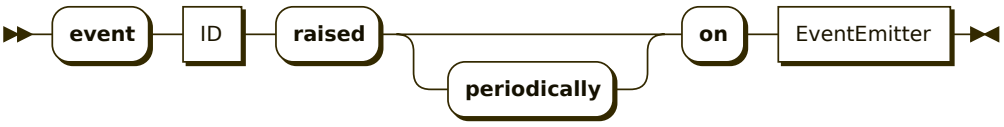
Listing 4.5 shows an example of a simple event handler with two parameters declared.

```
1 handler throughputCalc(IntervalEvent e, Map<String, Object> data) {
2    // logic to calculate throughput
3 }
```

Listing 4.5: *Example of event handler declaration*

Event declarations are specified with a name, the optional keyword `periodically`, indicating whether or not the event is time-based, and an event emitter. In case the event is time-based, the event emitter must resolve to a chronological expression that is based on the cron Unix scheduler, with the difference that PASCANI allows scheduling events in a resolution of seconds. On the contrary case, the event may be a service execution event, or a variable change event.

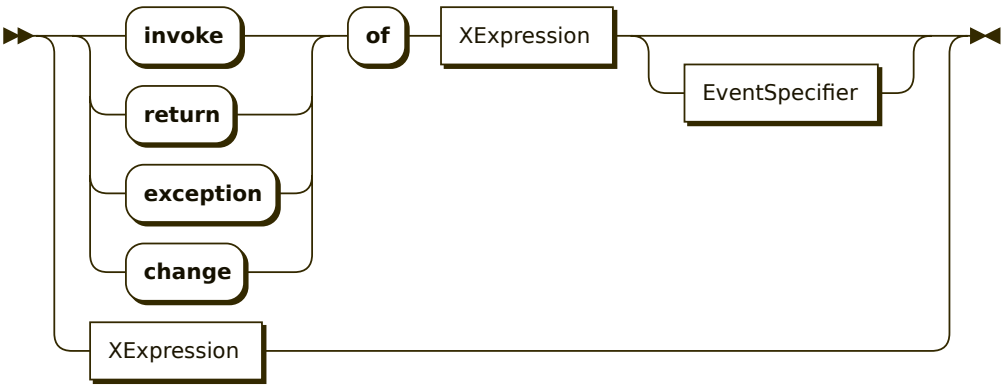
EventDeclaration:



Event emitters are specified according to the event type. For execution events, the emitter, also called the *target*, is an FPath expression [14] pointing to the SCA component, service, or reference to intercept. When the target is a component, all of its services and references are intercepted. For change events, the emitter is a variable from a namespace. In this case the emitter is specified using the variable accessor.

For change events, there is an additional (optional) parameter called the *event specifier*. The event specifier allows putting logical conditions on the new value to determine whether the subscribed handlers should be notified or not. This is only applicable for numerical variables.

EventEmitter:



Listing 4.6 shows some examples on different event declarations.

```
event e1 raised periodically on `*/5 * * * * ?` // every 5 seconds
event e2 raised on invoke of "$domain/scachild::Server/scaservice::print"
event e3 raised on change of SLI.Performance.Throughput.actual
event e4 raised on change of SLI.Performance.Throughput.actual below 80
event e5 raised on change of System.Performance.ResponseTime above 1.5 or below 0.5
event e6 raised on change of System.Performance.ResponseTime equal to 0.5
```

Listing 4.6: *Example of event declarations*

Variable declarations inside monitors are specified in the same way that context variables in namespaces (*cf.* Subsection 4.4.2).

The last statement is the configuration block. It is intended to either *i)* subscribe event handlers to events, *ii)* specify a URI when the event emitter is not deployed in the same monitor's host or specify a port different than the default one, or *iii)* indicate that a probe should be also introduced in the specified event emitter. As configuration blocks are executed after monitor instantiation, it is also useful for initialiazing the monitor variables that require more complex initialization than a single expression.

EventEmitter:

Listings 4.7 and 4.8 show two PASCANI specifications for monitoring throughput and response time of an SCA service. For sake of simplicity, these examples only include updating the context variables, not reacting to changes on them. Reacting to contextual changes would require declaring a *change* event on the variable of interest, and following the same subscription pattern described in the aforementioned listings.

```
package com.company.monitoring

namespace SystemVars {
   var throughput = 0d
   var latency = 0d
}
```

Listing 4.7: *Namespace specification for monitoring Throughput and Response time*

```
package com.company.monitoring
```

```
2
3  import java.util.Map
4  import org.pascani.dsl.lib.Probe
5  import org.pascani.dsl.lib.events.IntervalEvent
6  import org.pascani.dsl.lib.events.ReturnEvent
7
8  using com.company.monitoring.SystemVars
9
10 monitor Performance {
11
12    val searchTags = #{ "service" -> "search", "host" -> "grid0" }
13    val paymentTags = #{ "service" -> "payment", "host" -> "grid1" }
14    val server = "$domain/scachild::Server"
15
16    event minutely raised periodically on `0 * * * * ?`
17    event search raised on return of server + "/scaservice::search"
18    event payment raised on return of server + "/scaservice::payment"
19
20    handler updateLatency(ReturnEvent e, Map<String, Object> data) {
21       SystemVars.latency = tag(e.value, data.mapValues[v|String.valueOf(v)])
22    }
23
24    handler updateTroughput(IntervalEvent e) {
25       val now = System.currentTimeMillis()
26       val searchCount = search.probe.countAndClean(-1, now)
27       val paymentCount = payment.probe.countAndClean(-1, now)
28       SystemVars.throughput = tag(searchCount, searchTags)
29       SystemVars.throughput = tag(paymentCount, paymentTags)
30    }
31
32    config {
33       #[ search, payment ].map[e | e.useProbe = true]
34       search.bindingUri = new URI("http://localhost:5000")
35       payment.bindingUri = new URI("http://localhost:5001")
36       search.subscribe(updateLatency, searchTags)
37       payment.subscribe(updateLatency, paymentTags)
38       minutely.subscribe(updateThroughput)
39    }
40 }
```

Listing 4.8: *Monitor specification for monitoring Throughput and Response time*

Listing 4.7 depicts the `SystemVars` namespace declaring two context variables: `thoughput`

and `latency`, both of type `double`. This namespace is used by the `Performance` monitor presented in listing 4.8. Lines 3 to 6 of listing 4.8 contain Java imports used in the rest of the specification. Line 8 declares that `Latency` makes use of `SystemVars`. Lines 16 to 18 declare a time-based event, and two execution events on the `search` and `payment` services. Line 33 indicates that the execution events `search` and `payment` must introduce a probe into the corresponding service; without this, there would not be way of pulling measurement data to calculate service throughput. Lines 34 and 35 tell PASCANI where is the component running, in order to introduce the monitor probe. Lastly, lines 36 to 38 subscribe each handler to the corresponding event.

From monitor specifications, PASCANI generates the elements composing the dynamic monitoring infrastructure. It also generates the corresponding deployment specifications. These specifications are written in AMELIA, and follow the syntax and semantics explained in the next section.

4.5 AMELIA: A DSL for Dynamic Software Deployment

Amelia is a declarative and rule-based Domain Specific Language for automating the deployment of distributed component-based software systems. It is inspired in tools such as Ant[1] and Maven[2], although its syntax is based on the Make[3] build automation tool, providing commands to facilitate the execution of deployment tasks across multiple computing nodes. Its expressions and statements are based on the Xbase expression language [14].

From AMELIA specifications, the language implementation generates executable deployment artifacts that perform the tasks required to transfer, install, and configure the software components to deploy on each of the specified processing nodes, using the SSH and SFTP protocols, and executing the commands specified in AMELIA rules.

4.5.1 Language Concepts

The AMELIA language consists of two main elements: subsystems and deployments. A subsystem contains a set of deployment operations for a self-contained system that belongs to a larger system. A deployment contains flow control statements that execute the deployment a set of subsystems in a particular way.

Subsystems are made of execution rules that are executed into specific computing nodes. Said rules are dependable containers of commands, that guide the deployment of software components.

[1] http://ant.apache.org/
[2] https://maven.apache.org/
[3] https://www.gnu.org/software/make/

4.5.2 The AMELIA Grammar Definition

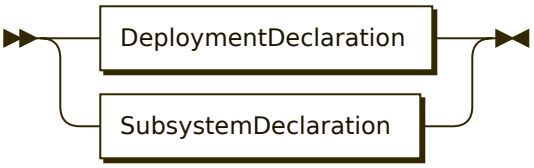
In this subsection we present the AMELIA grammar definition, introduce the main language constructs and their associated semantics using syntax diagrams. Syntax diagrams, or railroad diagrams, are a visual way to represent context-free grammars. Each diagram defines a non-terminal. A diagram describes possible paths between an entry point and an end point, by going through other non-terminals and terminals. Terminals are represented with round boxes, while non-terminals are represented with square boxes. The complete grammar is presented in Appendix D. Grammar rules starting with an X are inherited from Xbase unless they are explicitly re-defined within the grammar.

A valid specification file in AMELIA has a mandatory definition of a package name, and optional definition of a Java import section and a type declaration. This means that files without a type declaration are valid specifications that do not generate any artifact. A compilation unit is a translation unit, that is, a source file that contains the definition of either a Subsystem or a Deployment.

DeploymentSpecificationModel:

Subsystems contain the definition of hosts (*i.e.*, computing nodes) and execution rules; and Deployments allow configuring deployment strategies from subsystem declarations.

TypeDeclaration (Compilation unit):

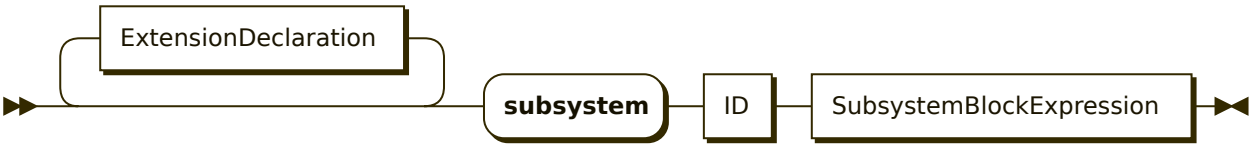


A deployment declaration provides the means to perform custom deployment behavior, such as systematically repeating the same deployment a given number of times, or retrying on failure. This type of declaration is composed of an optional extension section, a name, and a block of expressions.

DeploymentDeclaration:

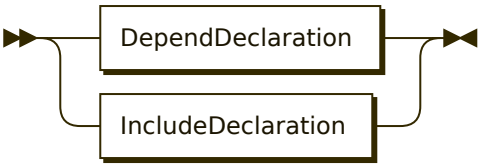
A subsystem is a composable and dependable unit of deployment intended to specify how to deploy a set of components into different hosts. Its declaration contains an optional extension section, a name, and a block of expressions.

SubsystemDeclaration:



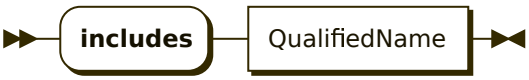
Extension declarations are extensions for either including a subsystem, or specifying an execution dependency. On the one hand, including a subsystem into another subsystem means all of the included subsystem's parameters and execution rules are *inserted* and can be treated as part of the subsystem. Name clashes are handled by making the colliding parameters accessible using its qualified name. On the other hand, when a subsystem is included into a deployment declaration, it is taken into account in the deployment strategy, allowing to instantiate the subsystem using different values for its parameters.

ExtensionDeclaration:



In order to include a subsystem, the `include` declaration must specify the qualified name of the included subsystem, that is, its package name concatenated with its name, with a dot separating every word.

IncludeDeclaration:



Subsystem dependencies are a way to establish execution order to ensure component dependencies. Specifying a dependency requires explicitly declaring it with the qualified name of the deployment or subsystem declaration.

DependDeclaration:

Listings 4.9 and 4.10 show the contents of two valid files written in Amelia given the grammar rules above.

```
1 package com.company
2
3 includes Common
4 depends on Test1
5
6 subsystem Test2 {
7    // Variables and on-host declarations
8 }
```

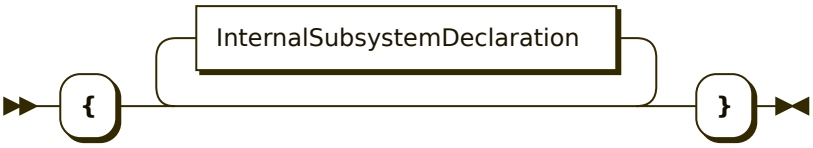
Listing 4.9: *Minimal example of a subsystem specification*

```
1 package com.company
2
3 includes Test1
4 includes Test2
5
6 deployment CustomStrategy {
7    // Deployment expressions
8 }
```

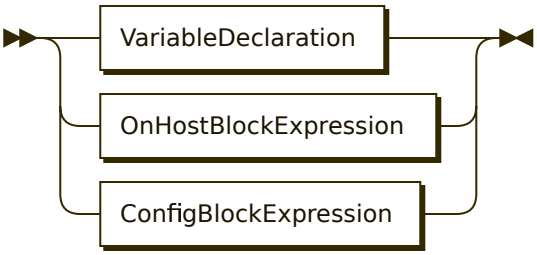
Listing 4.10: *Minimal example of a deployment specification*

The block of expressions composing a subsystem can contain variable declarations, on-host blocks of expressions, and configuration blocks.

SubsystemBlockExpression:

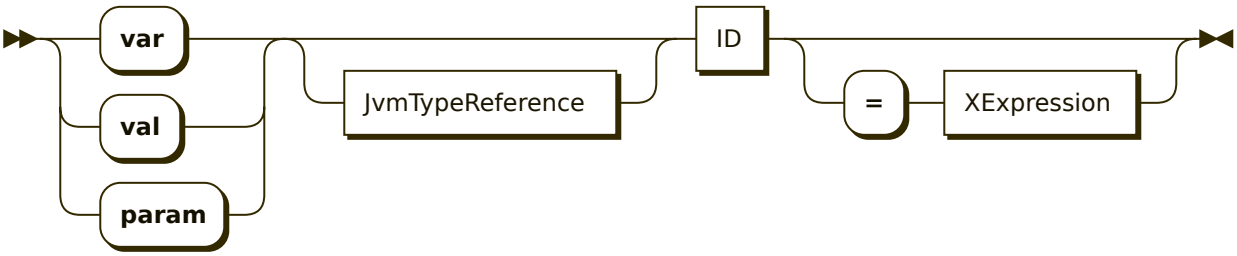


InternalSubsystemDeclaration:



In Amelia, a variable declaration can be either an immutable value, a variable, or a parameter. The first two cases refer to regular private variables, while the third one has different semantics associated. Parameters are included in the subsystem's constructor, meaning that subsystems can be instantiated with different arguments. When subsystems are included, the included parameters are also passed to the subsystem's constructor. Therefore, in an include chain, the leaf subsystem's constructor (*i.e.*, the last subsystem) would contain all of the parameters included in all of the subsystems in the include chain; the order is determined by the order in the include declarations.

VariableDeclaration:

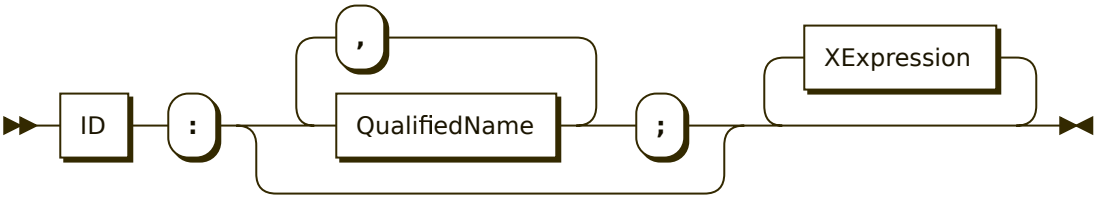


On-host blocks of expressions group rules to be executed in the given host.

ConfigBlockExpression:

Deployment actions are specified using *rules*. Each rule is composed of a target, an optional enumeration of dependencies (*i.e.*, other targets), and a set of commands. When a rule depends on other rules, it means its commands cannot be executed until all of the commands declared on the other rules are executed. Listing 4.11 shows an example of rule declaration and dependencies.

RuleDeclaration:




```
1 init:
2    // commands
3 server: init;
4    // commands
5 client: init, server;
6    // commands
```

Listing 4.11: *Example of declaration of execution rules*

Amelia Commands

In order to facilitate the specification of deployment tasks, Amelia provides a set of commands with automatic error checking during compilation and execution, and recognition of successful and erroneous states. The provided commands allow to *i)* change the working directory, *ii)* compile a FraSCAti component, *iii)* run a compiled component, *iv)* transfer files or directories to a remote processing node, *v)* evaluate FScript expressions on a given FraSCAti runtime, and *vi)* execute Unix-like commands.

In addition to this, Amelia allows configuring any predefined value for each command, in order to modify the command's behavior. Adding `...` to the end of a command declaration would make

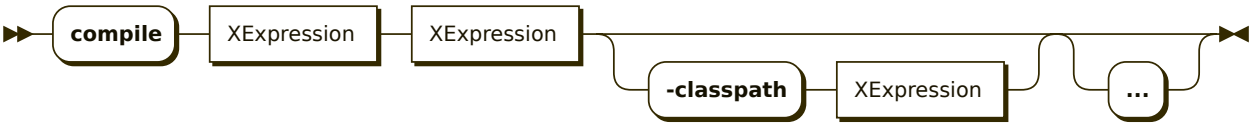
the command's builder accessible, which allows to modify its internal values. Listed below are the AMELIA commands:

The cd command requires only the new working directory, which must resolve to a string expression.

Cd:

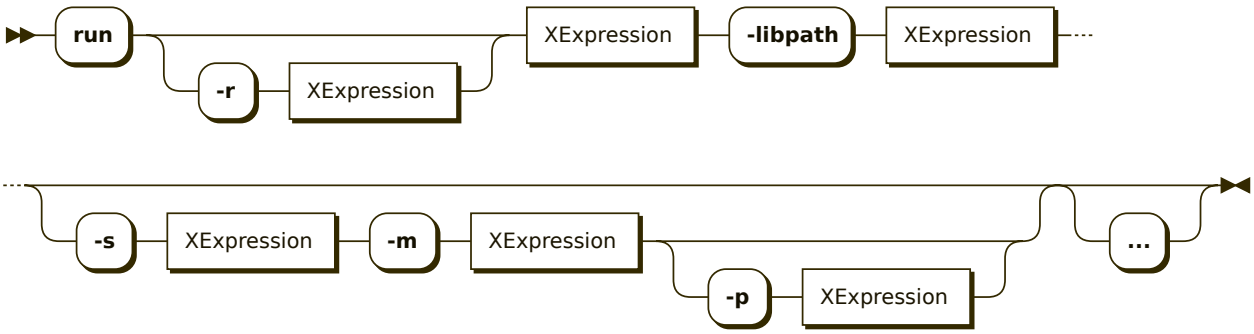
The compile command is composed of string expressions representing, from left to right, the source code directory, the output file, and optionally the classpath.

Compile:



The run command is composed of an optional integer expression, representing the port in which the FRASCATI FScript console is exposed, a string representing the component (*i.e.*, the .composite file), and the classpath. When the component is run in client mode, additional parameters must be specified, including the sevice name, method name, and optionally a list of arguments, all of them being string expressions.

Run:

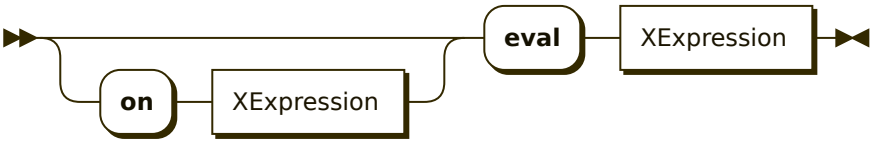


The transfer command requires the local and remote location of either a file or directory. If the remote location does not exist, it will be created.

Transfer:

The eval command optionally expects the URI where the FraSCAti runtime is running, and the FScript script to evaluate.

Eval:



Any other Unix-valid command is given as a string.

Custom:

Listing 4.12 shows several examples of command specifications.

```
1 cd "/home/user/projects"
2 compile "src" "project" -classpath #[ "libs/lib1.jar", "lib/lib2.jar" ]
3 run -r 5000 "server" -libpath #[ "server.jar", "lib/common.jar" ]
4 run "client" -libpath #[ "client.jar", "lib/common.jar" ] -s "r" -m "run"
5 scp "~/project/files" to "/tmp/project/files"
6 on new URI("http://localhost:5000") eval "some-procedure()"
7 cmd "date >> date.txt"
```

Listing 4.12: *Examples of command specifications*

The *helloworld-rmi* Example

Listings 4.13 and 4.14 show the deployment specification for the *helloworld-rmi* example that comes with the FraSCAti distribution. This example comprises two SCA components: a `server`, exposing a print service through RMI, and a `client`, consuming that service to print a message in the standard output. Notice that `$FRASCATI_HOME` is not related to Amelia in any way, it is just an environment variable that gets resolved during the SSH session.

The `WarmingUp` deployment specification (cf. line 8 of Listing 4.14 combines two common strategies when deploying systems: executing the system several times in order to prepare the environment to execute performance tests, and retrying the deployment if a failure is identified. The first strategy is specified by means of the `start` method and the `for` statement in line 14. The start method can be invoked with two parameters, the first one indicates whether or not the executed components must be stopped after deployment, and the second one indicates whether or not a shutdown must be performed. Turning off the deployment shutdown if useful when the user wants to observe the SSH session's output (*i.e.*, the standard output of the executed components). The second strategy is realized by means of the `RetryableDeployment` utility, which re-executes the given lambda function the number of times indicated by the second parameter. In this case, as noted in line 16, the deployment would be executed one time, and depending on its result, successful or not, the utility would try again two more times.

By default, subsystems are initialized using an empty constructor; in case there are uninitialized parameters, an instance of the subsystem must be provided to avoid errors associated with invocations on `null` objects. Line 12 shows how to set a subsystem's instance using a different constructor. This can also be used to provide different subsystem instances per deployment; in this case, moving the `set` invocation inside the for statement would allow to deploy the system into a different host in each iteration, assuming a different host object is passed to the subsystem's constructor.

```
1  package com.company
2
3  import org.amelia.dsl.lib.descriptors.Host
4
5  subsystem Helloworld {
6
7     param Host host = new Host("localhost", 21, 22, "username", "password")
8
9     on host {
10       compilation:
```

```
11          cd "$FRASCATI_HOME/examples/helloworld-rmi/"
12          compile "server/src" "s"
13          compile "client/src" "c"
14
15       execution: compilation;
16          run "helloworld-rmi-server" -libpath "s.jar"
17          run "helloworld-rmi-client" -libpath "c.jar" -s "r" -m "run"
18    }
19
20 }
```

Listing 4.13: *Subsystem specification for the helloworld-rmi example*

```
1  package com.company
2
3  import org.amelia.dsl.lib.util.RetryableDeployment
4  import org.amelia.dsl.lib.descriptors.Host
5
6  includes com.company.Helloworld
7
8  deployment WarmingUp {
9
10    val helper = new RetryableDeployment()
11    val remote = new Host("192.168.99.100", 25632, 41256, "username", "password")
12    set(new Helloworld(remote))
13
14    for (i : 1..10) {
15       helper.deploy([
16          start(true)
17       ], 3)
18    }
19
20 }
```

Listing 4.14: *Deployment specification for the helloworld-rmi example*

Chapter 5

Implementation

Pascani and Amelia were developed using the Java programming language, and the Xtext framework. Xtext is a language engineering framework that assists developers in the generation of language implementations, including parser, linker, type checker, and compiler, as well as edition support for well known IDEs such as Eclipse and IntelliJ, from a grammar definition. The complete grammar definitions of Pascani and Amelia are presented in Apendix C and Apendix D, respectively. To support the functionalities of each language, we developed a runtime library abstracting common elements used by all applications generated by the compiler. This allows each compiler to reduce the amount of lines of code generated and modularize the implementation in a way that new elements can be created by combining the existing ones. In the case of Pascani, there is an additional library containing resources and elements supporting all SCA-related concepts.

The current implementation of both languages has been customized for the Eclipse IDE to offer syntax-driven edition, static error checking, code refactoring, and code generation. To install Pascani or Amelia, it is enough to add the corresponding update site[1] and follow the installation instructions.

5.1 Pascani

The implementation code for the Pascani DSL comprises 11.028 SLOC (without counting generated source code) distributed along the projects listed in Table 5.2. This section presents an overview of the most relevant projects, including the implementation details of the language syntax and semantics for realizing the translation model used to implement the compiler, and the supporting libraries.

The language semantics are written using the facilities provided by Xtext, that is, checking methods that are invoked once the model has been parsed. As an example, listing 5.1 depicts

[1] `http://unicesi.github.io/pascani/releases` and `http://unicesi.github.io/amelia/releases`

Project	Description	SLOC
Eclipse Features		
org.pascani.dsl.feature	Contains the definition of the plug-in projects composing the PASCANI DSL.	785
org.pascani.dsl.build.feature	The PASCANI update site.	7
Eclipse Plug-ins		
org.pascani.dsl	This project contains the classes that belongs to the language core library. It contains the PASCANI compiler, code generator, JVM model, scope semantics, type system and semantic validation.	2257
org.pascani.dsl.ide	Contains the classes related to the Eclipse IDE (none, at the moment).	21
org.pascani.dsl.ui	Contains the plug-in definitions and corresponding Java implementation regarding the Eclipse user interface. It contains classes that implements or configures the IDE content assist, highlighting rules, editor, labeling, outline, and quick fix.	677
org.pascani.dsl.lib.osgi	Wrappers the PASCANI runtime libraries into an OSGi bundle.	13
Maven Projects		
org.pascani.dsl.lib	Contains all of the classes supporting the components of the Dynamic Monitoring Infrastructure that are generated by the compiler.	2666
org.pascani.dsl.lib.compiler	Contains utility classes used by the PASCANI compiler.	987
org.pascani.dsl.lib.sca	Contains all of the classes and resources supporting the runtime activity related to the SCA domain.	1887
org.pascani.dsl.dbmapper	Contains the classes that map monitoring events into plain data that can be persisted in a database.	1128
org.pascani.dsl.target	Contains the Eclipse target definition.	15
org.pascani.tycho.parent	Parent pom that configures the build life-cycle of the plug-in and maven projects.	239
Web Projects		
org.pascani.dsl.web	Contains classes and resources to publish a web editor using the PASCANI compiler and the Eclipse UI plug-ins as services.	346

Table 5.2: *Java projects composing the implementation of* PASCANI

a semantic rule for finding invalid parameter types in event handlers. The semantic validation performed in this method is as follows: event handlers cannot have more than two parameters, and their type must be Event, and Map with type parameters String and Object, respectively.

```
1  @Check
2  def checkHandlerParameters(Handler handler) {
3     if (handler.params.size > 2) {
4        error("Event handlers cannot have more than two parameters",
             PascaniPackage.Literals.HANDLER__PARAMS)
5     }
6
7     if (handler.params.get(0).actualType.getSuperType(org.pascani.dsl.lib.Event) ==
          null) {
8        error('''TheÂńIF handler.params.size > 1 firstÂńENDIF parameter must be subclass
             of Event''', PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
9     }
10
11    if (handler.params.size > 1) {
12       val actualType = handler.params.get(1).actualType.getSuperType(Map)
13       val showError = actualType == null
14          || actualType.typeArguments.size != 2
15          || !actualType.typeArguments.get(0).identifier.equals(String.canonicalName)
16          || !actualType.typeArguments.get(1).identifier.equals(Object.canonicalName)
17       if (showError)
18          error('''The second parameter must be of type Map<String, Object>''',
                PascaniPackage.Literals.HANDLER__NAME, INVALID_PARAMETER_TYPE)
19    }
20 }
```

Listing 5.1: *Semantic rule for finding invalid parameters in Event Handlers (written in the Xtend programming language)*

5.1.1 The Pascani Runtime Library

The Pascani runtime library contains all of the classes supporting the components of the Dynamic Monitoring Infrastructure that are generated by the compiler. These classes are organized into seven packages. Figure 5.1 depicts a simplified class diagram containing all of the classes inside each package. We describe the contents of each package as follows:

1. org.pascani.dsl.lib

Contains the classes derived from the elements of the Dynamic Monitoring Infrastructure in its basic form (*i.e.*, interfaces or abstract classes).

2. org.pascani.dsl.lib.events
Contains the event types supported by the infrastructure; that is, all subtypes of `Event`. However, the language supports only time-based events (*i.e.*, `IntervalEvent`), variable change events (*i.e.*, `ChangeEvent`), and execution events (*i.e.*, `InvokeEvent`, `ReturnEvent`, `TimeLapseEvent`, and `ExceptionEvent`).

3. org.pascani.dsl.lib.infrastructure
Contains the classes realizing the monitoring infrastructure, such as implementations of `Namespace` and `Probe`, and the basic definitions of the element participating in the communication mechanisms (*e.g.*, Consumer, Producer, RPC server and client).

4. org.pascani.dsl.lib.infrastructure.rabbitmq
Contains implementations of the elements participating in the communication mechanisms. For implementing the `Message Broker` component presented in Section 4.3, and the RPC server/clients scheme, we decided to use RabbitMQ[2].

5. org.pascani.dsl.lib.util
Contains both utility classes used by other classes, and classes to work with sets of events.

6. org.pascani.dsl.lib.util.events
Contains classes to realize handlers, events, and event subscriptions.

7. org.pascani.dsl.lib.util.log4j
Contains utility classes for appending logs to the infrastructure.

5.1.2 The Pascani SCA Library

The Pascani SCA library contains all of the classes and resources supporting the runtime activity related to the SCA domain. The elements composing this library are: *i)* a set of interceptors configured to produce the various types of execution events supported by the language, *ii)* FScript procedures to add and remove monitor probes at runtime, and *iii)* facilities to ease the introspection of FraSCAti applications.

Figure 5.2 depicts a simplified class diagram of the Pascani SCA library. Classes within this library are organized into the following packages:

1. org.pascani.dsl.lib.sca
Contains utility classes to introspect FraSCAti applications and manipulate monitor probes at runtime.

[2] `https://www.rabbitmq.com/`

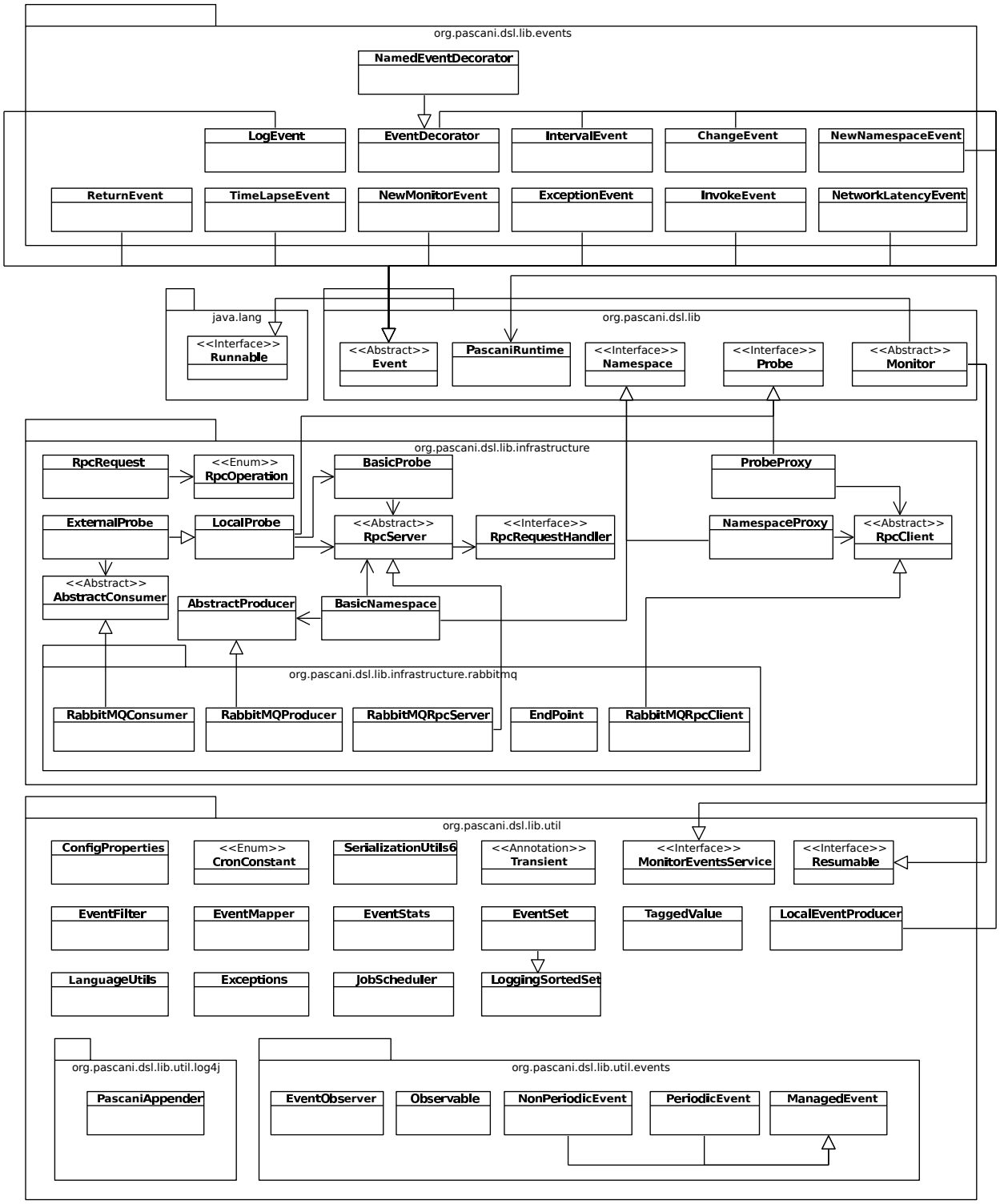


Figure 5.1: *Simplified Class Diagram of the* Pascani *Runtime Library*

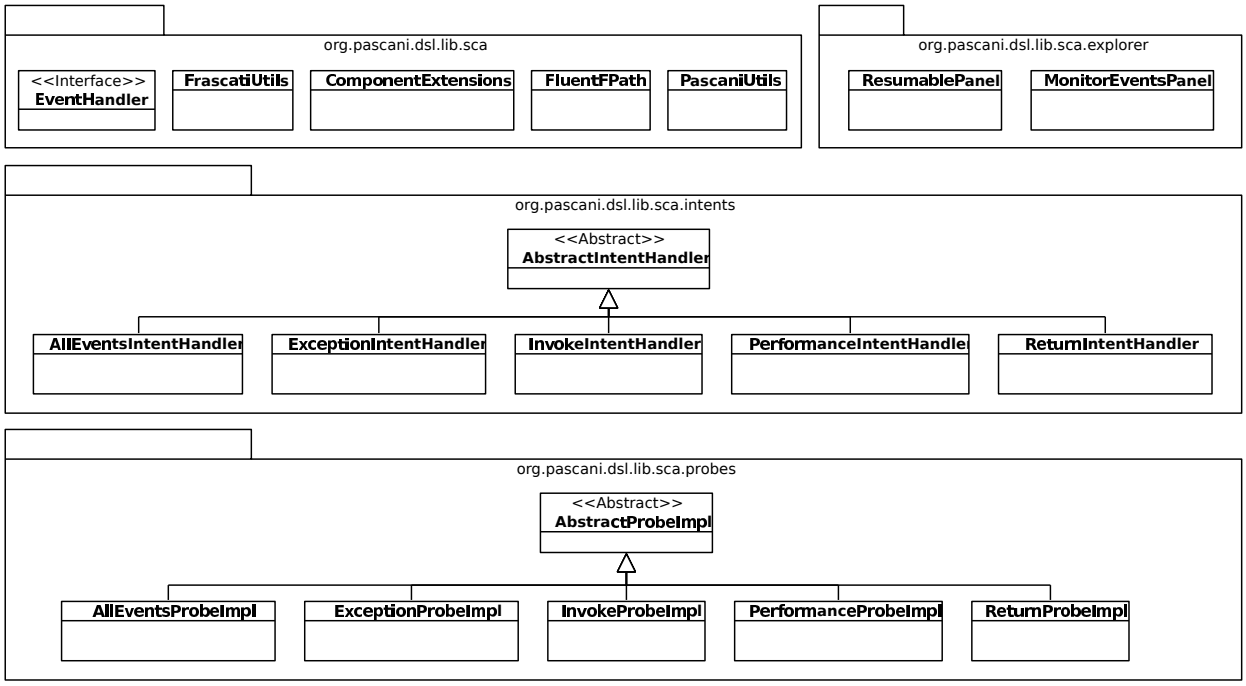


Figure 5.2: *Simplified Class Diagram of the* Pascani *SCA Library*

2. org.pascani.dsl.lib.sca.explorer
Contains UI extensions to the FraSCAti Explorer that allows managing monitors's state and their events.

3. org.pascani.dsl.lib.sca.intents
Contains the implementation of service interceptors according to the execution events supported by the language.

4. org.pascani.dsl.lib.sca.probes
Contains the implementation of the monitor probes configured to handle the execution events supported by the language.

5.1.3 The Pascani Translation Model

The Pascani compiler translates Namespace and Monitor specifications into SCA components, with a Java implementation. The generated Java classes use the elements from the runtime and SCA libraries, which considerably reduces the amount of lines of code generated. The following subsections detail the mapping between elements from Pascani specifications and their corresponding Java class elements.

Namespace Declarations

From a namespace specification, the compiler generates one composite file (*i.e.*, an SCA component descriptor) and two Java classes: a namespace implementation, which is also the implementation of the generated SCA component, and a namespace proxy. The first one is a realization of the `Namespace` interface we proposed in Section 4.3, with all of the namespace's variables registered. The second one is a proxy class mediating between client elements and the namespace implementation. For instance, when a monitor reads a context variable's value, the request passes through the proxy. Then it creates an RPC request and send it to the namespace implementation. This then answers the RPC request, and the proxy returns the variable value to the monitor.

Figures 5.3 and 5.4 depict an abstract view of the mapping between namespaces and Java classes elements. From a namespace specification, the compiler uses the package, name, import section, and documentation for generating both the namespace implementation and proxy.

On the one hand, a generated namespace implementation is very simple, as it inherits from the `BasicNamespace` class that already implements the behavior described in Section 4.3. Each of the variables declared within the namespace, including the ones declared within internal namespaces, is translated into a registration statement, that is, a method invocation.

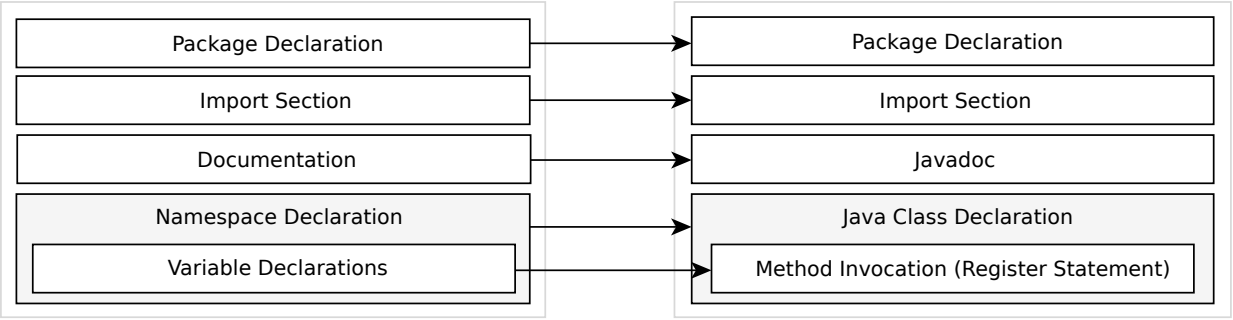


Figure 5.3: *Mapping between a Namespace definition (left) and its corresponding Java class elements (right)*

On the other hand, a generated namespace proxy is a more complex yet simple class. Variable declarations are translated into getter and setter methods, including variants for allowing reading and updating values having into account contextual information. Each method forwards the invocation to equivalent methods on the class `NamespaceProxy`. Internal namespace declarations have the same treatment, with the difference that they are turned into private fields of the container namespace. The generated class is a utility for simplifying the notation in the language specification. As Pascani is based on the Xbase expression language, regular invocations like `object.method(parameter)` can be re-written as an assignment. In this way, `SLI.latency = 0.5`

is equivalent to `SLI.latency(0.5)`. In the same manner, invocations with no parameters can be written without parenthesis, such as `SLI.latency`. In summary, the generated namespace proxy acts as a utility declaring the method hierarchy representing the namespace and variables hierarchy.

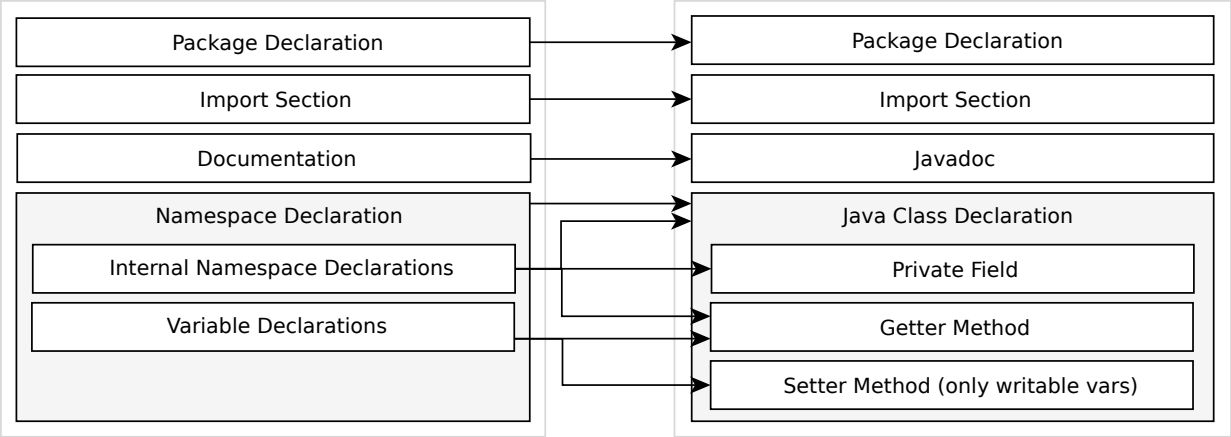


Figure 5.4: *Mapping between a Namespace definition (left) and its corresponding Java -Proxy- class elements (right)*

Monitor Declarations

From a monitor declaration, the PASCANI compiler generates a composite file (*i.e.*, SCA component descriptor) along with its corresponding Java implementation. As for namespace declarations, the generated Java class uses the package, name, import section, and documentation as they are declared in the specification file. Namespace uses and variable declarations are translated into static private fields, making them accessible from the entire monitor implementation, including inner classes. Event declarations are also translated into fields; their type can be either `PeriodicEvent` or `NonPeriodicEvent`. In the latter case, an inner class is created since it requires an implementation. From event handlers, the compiler generates both a private field and an inner class inheriting from `EventObserver`. Lastly, configuration blocks are translated into instance methods that are invoked sequentially according to its order of appearance in the monitor specification.

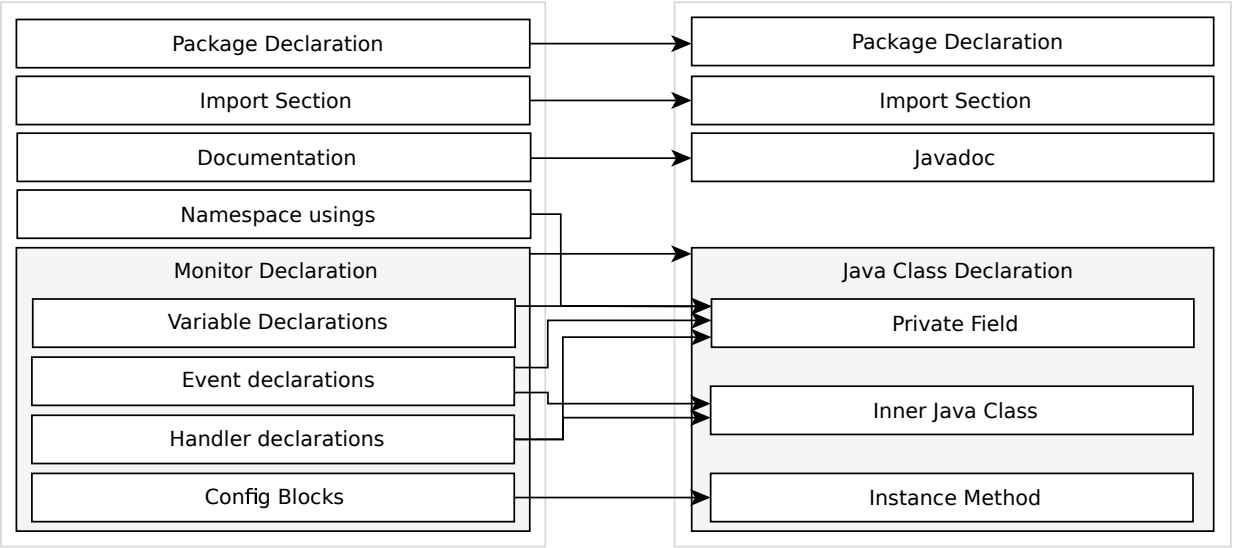


Figure 5.5: *Mapping between a Monitor definition (left) and its corresponding Java class elements (right)*

5.1.4 Storage and Visualization of the Monitoring Data

The Dynamic Monitoring Infrastructure supports several technologies for persisting the context variables, and for visualizing them as well. By default, PASCANI supports storing data in InfluxDB[3], ElasticSearch[4], RethinkDB[5], FnordMetric [6], and CSV files. From these files, different open source data visualization products such as Grafana[7], FnordMetric, and Kibana[4] can be used for visualizing graphical representations of the monitoring data.

Figure 5.6 depicts the deployment diagram of the monitoring infrastructure generated by the PASCANI compiler. Monitor and namespace specifications, as well as the elements of the target system, are compiled to `jar` files, that are deployed to a FRASCATI execution environment. Both artifacts monitors and namespaces depend upon the PASCANI runtime libraries, therefore they must be deployed jointly. The datastore mapper component is not an SCA component, that is why it only requires a Java execution environment. This component is subscribed to namespaces using the RabbitMQ message broker.

[3] `https://influxdata.com/`
[4] `https://www.elastic.co/`
[5] `https://www.rethinkdb.com/`
[6] `http://fnordmetric.io/`
[7] `http://grafana.org/`

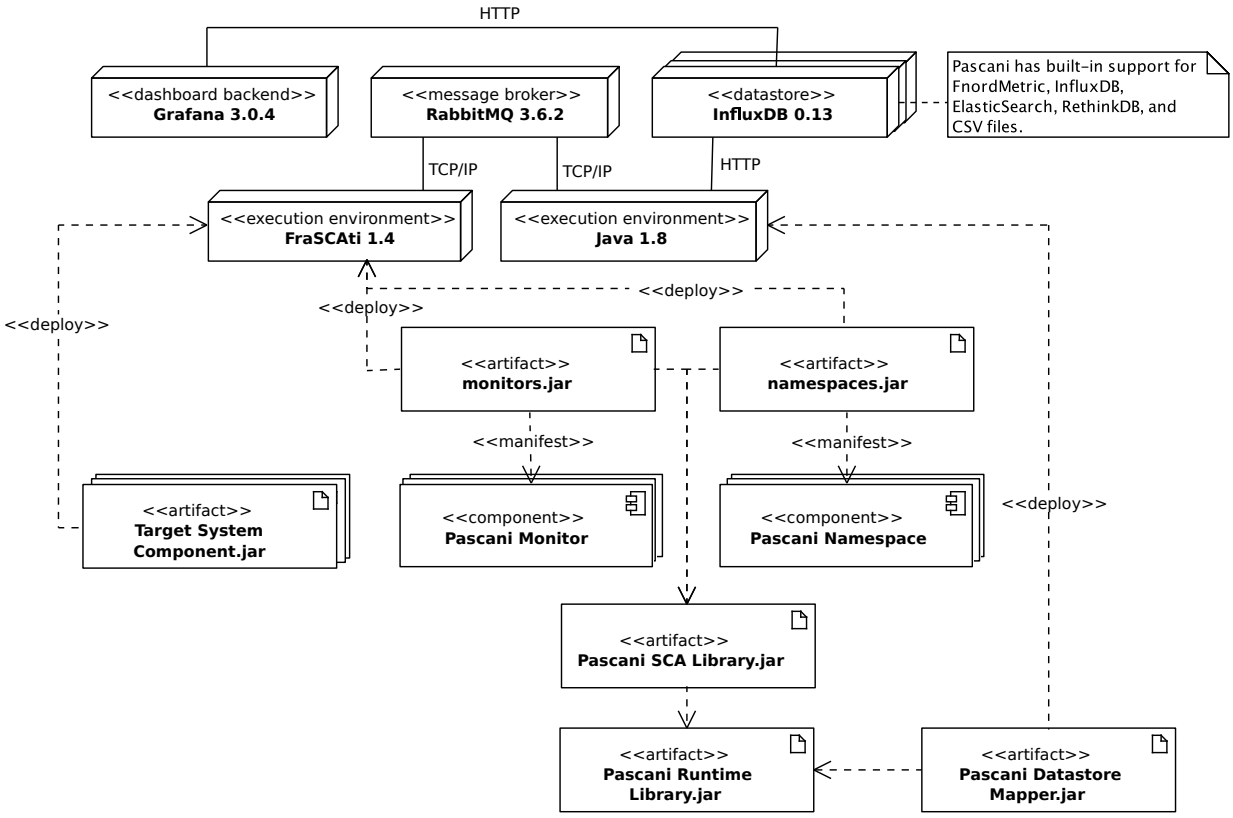


Figure 5.6: *Deployment Diagram of the Dynamic Monitoring Infrastructure*

5.2 Amelia

The implementation code for the Amelia DSL comprises 7,239 SLOC (without counting generated source code), distributed along the projects listed in Table 5.4. This section presents an overview of the most relevant projects, including the implementation details of the language syntax and semantics for realizing the translation model used to implement the compiler, and the supporting libraries.

The language semantics are written using the facilities provided by Xtext, that is, checking methods that are invoked once the model has been parsed. As an example, listing 5.2 depicts a semantic rule for validating the host parameter type in on-host expressions. The semantic validation performed in this method is as follows: if the actual (inferred) type of the expression is not of type `Host` or an `Iterable` of `Host` objects, show an error.

```
1 @Check
2 def void checkHost(OnHostBlockExpression blockExpression) {
3    val type = blockExpression.hosts.actualType
4    val isOk = type.getSuperType(Host) != null || type.getSuperType(Iterable) != null
```

Project	Description	SLOC
Eclipse Features		
org.amelia.dsl.feature	Contains the definition of the plug-in projects composing the AMELIA DSL.	782
org.amelia.dsl.build.feature	The AMELIA update site.	7
Eclipse Plug-ins		
org.amelia.dsl	This project contains the classes that belongs to the language core library. It contains the AMELIA compiler, code generator, JVM model, scope semantics, type system and semantic validation.	1970
org.amelia.dsl.ide	Contains the classes related to the Eclipse IDE (none, at the moment).	21
org.amelia.dsl.ui	Contains the plug-in definitions and corresponding Java implementation regarding the Eclipse user interface. It contains classes that implements or configures the IDE content assist, highlighting rules, editor, labeling, outline, and quick fix.	603
org.amelia.dsl.lib.osgi	Wrappers the AMELIA runtime library into an OSGi bundle.	13
Maven Projects		
org.amelia.dsl.lib	Contains all of the classes supporting SSH and FTP session handling, execution scheduling and dependencies management, as well as logging and reporting functionalities.	3223
org.amelia.dsl.target	Contains the Eclipse target definition.	15
org.amelia.tycho.parent	Parent pom that configures the build life-cycle of the plug-in and maven projects.	262
Web Projects		
org.amelia.dsl.web	Contains classes and resources to publish a web editor using the AMELIA compiler and the Eclipse UI plug-ins as services.	343

Table 5.4: *Java projects composing the implementation of* AMELIA

```
5     val msg = '''The hosts parameter must be of type ÂńHost.simpleName or
          Iterable<ÂńHost.simpleName>, Âńtype.simpleName was found instead'''
6     val showError = !isOk
7        || type.getSuperType(List).typeArguments.length == 0
8        ||
             !type.getSuperType(Iterable).typeArguments.get(0).identifier.equals(Host.canonicalName)
9     if (showError) {
10       error(msg, AmeliaPackage.Literals.ON_HOST_BLOCK_EXPRESSION__HOSTS,
             INVALID_PARAMETER_TYPE)
11    }
12 }
```

Listing 5.2: *Semantic rule for validating the host parameter type in On-host expressions (written in the Xtend programming language)*

5.2.1 The Amelia Runtime Library

The Amelia runtime library is composed of classes implementing all of the concepts within the language, such as subsystems, commands, and dependencies. The most relevant subcomponents in this library are related to *i)* SSH and FTP session handling, *ii)* execution scheduling and dependencies management, *iii)* commands, and *iv)* logging and reporting functionalities.

Figure 5.7 depicts a simplified class diagram of the Amelia runtime library. Classes within this library are organized into the following packages:

1. org.amelia.dsl.lib
Contains the classes implementing SSH and FTP session handling, and task scheduling and execution.

2. org.amelia.dsl.lib.descriptors
Contains classes mostly used to describe commands, asset bundles, and hosts.

3. org.amelia.dsl.lib.util
Contains utility and logging classes.

5.2.2 The Amelia Translation Model

The Amelia compiler translates Subsystem and Deployment specifications into Java classes. These classes are entirely supported by the runtime library, and no other classes are generated. This means that the generated code has been reduced to the minimum, promoting reusability of the library's elements. The following subsections detail the mapping between elements from Amelia specifications and Java classes.

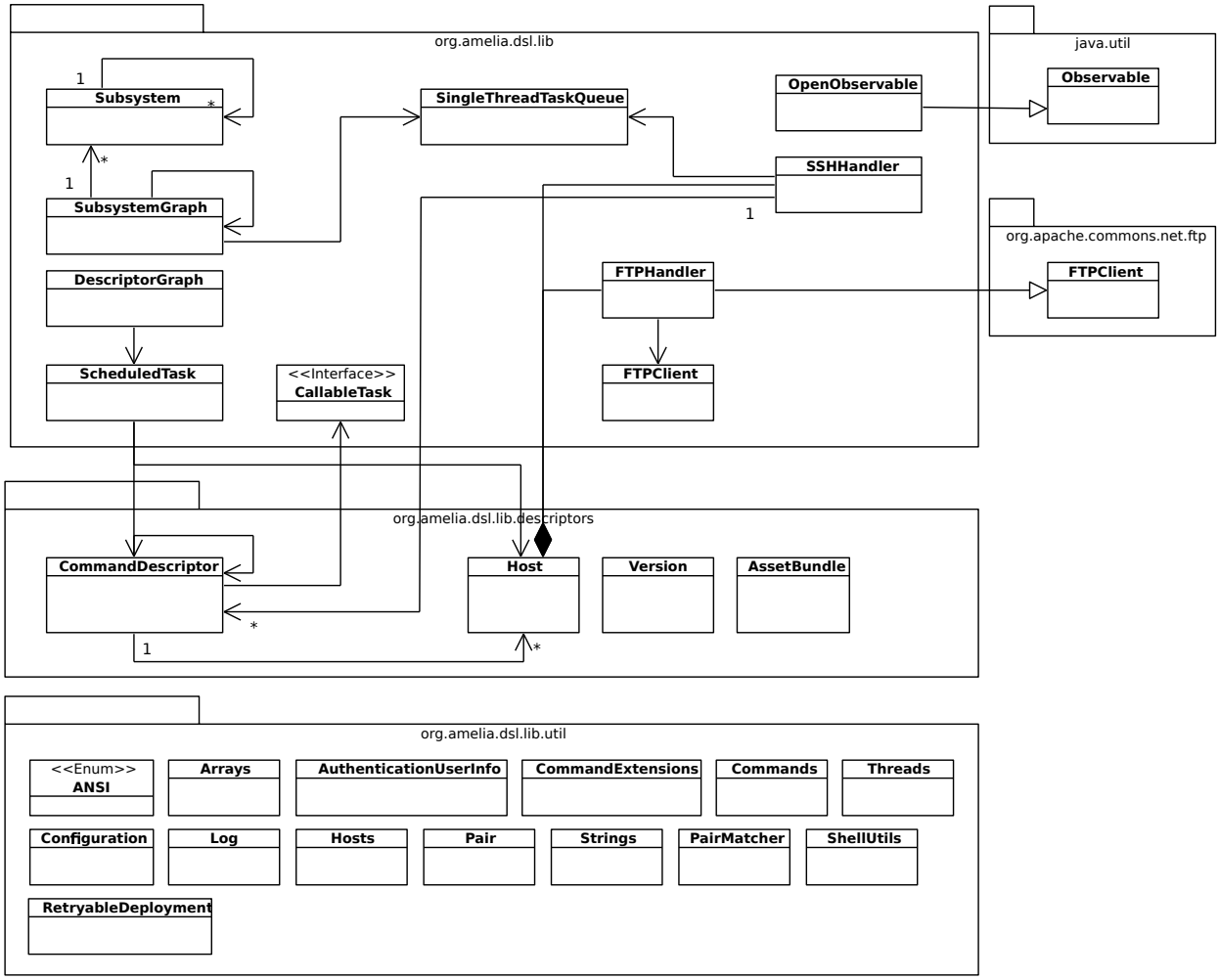


Figure 5.7: *Simplified Class Diagram of the* Amelia *Runtime Library*

Subsystem Specifications

Figure 5.8 depicts an abstract view of the mapping between elements from a subsystem specification and its corresponding derived Java class. From a given subsystem, the compiler generates a Java class using the subsystem's package and name. This class imports the Java classes specified in the subsystem's import section, and the runtime library's classes used in the rest of the generated code. Despite the resulting Java application not being thought to be analyzed by a human developer, the compiler generates readable code, including the available documentation in the Amelia's subsystem specification.

A subsystem inclusion is translated into a private field, whose type is of the included subsystem's derived class. This field is used for accessing the included parameters and execution rules. A subsystem dependency, as opposite to a subsystem inclusion, is not used in the subsystem's derived

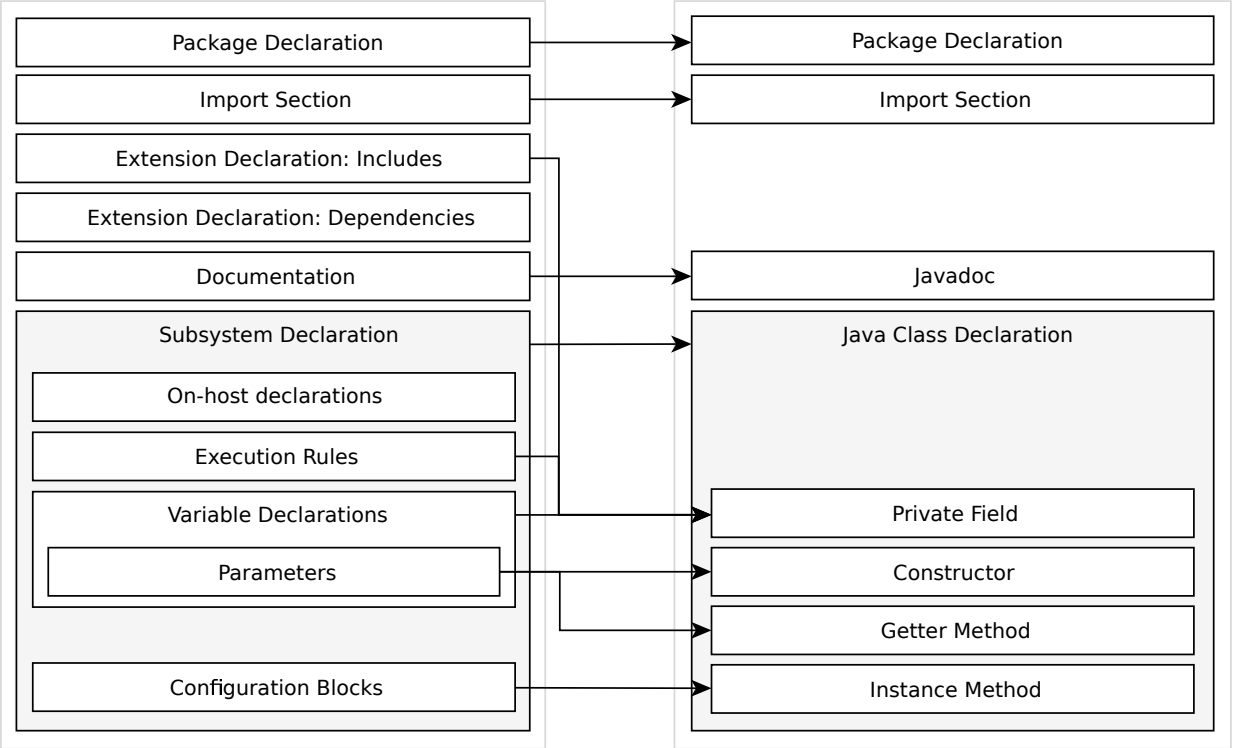


Figure 5.8: *Mapping between the Subsystems definition and the generated Java class elements*

class; dependencies are used only in deployments' derived classes

An on-host declaration is not translated into a specific Java element, but used to configure the commands inside of its execution rules; that is, for each command is given a host where it should be executed. Execution rules are translated into private array fields. Each field is initialized containing a set of `CommandDescriptor` instances, representing each command within the rule.

Inside subsystems there are two types of dependencies: sequential commands, and rule dependencies. The first type of dependency is represented in Java by configuring command $n + 1$ as a dependency of command n, where both commands are elements of the same execution rule. The second type of dependency is solved by configuring the last command of each dependency rule as dependency of the first command of the dependent rule. For instance, in listing 4.13, the first command of rule `execution` depends on the last command of rule `compilation`.

Variable declarations are translated into private fields. In case there are either subsystem parameters declared or included, a constructor is added to the generated class including them as parameters. A getter method is also created for each parameter.

Configuration blocks are limited to 1 per subsystem, as there is no reason to have more. If one is

declared, it is translated to an instance method. During execution, it is executed when all variables have been initialized and all commands have been configured.

Deployment Specifications

Figure 5.9 depicts an abstract view of the mapping between elements from a deployment specification and its corresponding generated Java class. As for subsystems, the compiler generates a Java class using the deployment's package, name and import section. A subsystem include is used to initialize a `Subsystem` instance using by default the subsystem's empty constructor. It is also used to establish subsystem dependencies when configuring the execution graph (*i.e.*, a `SubsystemGraph` instance). The subsystem's body is translated to an instance method as it is, and is invoked after the initialization of all subsystem instances.

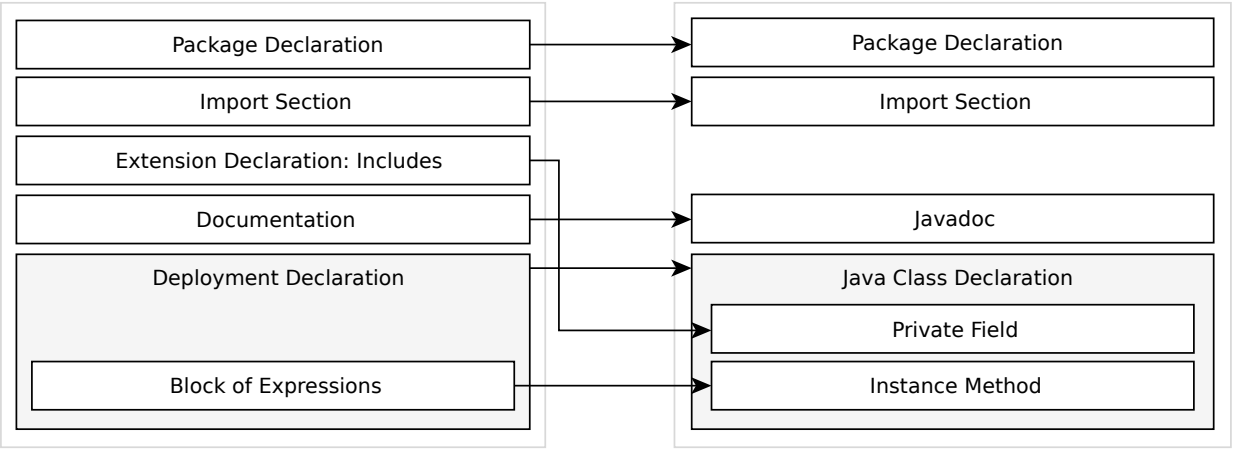


Figure 5.9: *Mapping between Deployments and Java classes' elements*

Chapter 6

Evaluation

Our solution consists of two main elements: the design of an scalable architecture for dynamic performance monitoring, and the design and implementation of Pascani and Amelia, two domain-specific languages for generating composable, traceable, and controllable monitoring components, and deploying them into a target system's running infrastructure, respectively. In this chapter we present a qualitative assessment of the effectiveness of Pascani and Amelia, based on real users' feedback. Included in the effectiveness evaluation, we also evaluated the applicability of the proposed architecture in a real scenario.

In order to assess the effectiveness of Pascani and Amelia, we use FQAD, a framework for qualitative assessment for DSLs proposed in [27]. FQAD refines a set of quality characteristics from the ISO/IEC 25010:2011 standard in order to use them in the assessment of DSLs. These characteristics are described as follows.

1. Functional suitability: the degree to which a DSL supports the development of solutions to satisfy stated requirements of the application domain.
2. Usability: the degree to which the DSL can be used by certain users to accomplish certain goals.
3. Reliability: the property of the language to help its users to produce reliable programs.
4. Maintainability: the degree to which the language promotes ease of program maintenance.
5. Productivity: the degree to which a language promotes programming productivity.
6. Extendability: the degree to which a language provides mechanisms for users to add new features.
7. Compatibility: the degree to which a DSL is compatible with the domain and development process.

8. Expressiveness: the degree to which a problem solution (in the domain) can be mapped into a program in the DSL naturally.

9. Reusability: the degree to which the language constructs can be used in other languages.

10. Integrability: the property of the language to be integrated with other languages used in the development process.

Figure 6.1 depicts the components within the FQAD Assessment Model, in which: *DSL success* is a group of interrelated characteristics in a DSL that collectively satisfies a requirement considered relevant for the DSL; *Goal statement* is intended to describe the purpose of the assessment; *DSL characteristics* are the characteristics described above, which are a set of unique attributes present in a high-quality DSL; *DSL sub-characteristics* are used to describe quality measures relevant to achieve the associated characteristic.

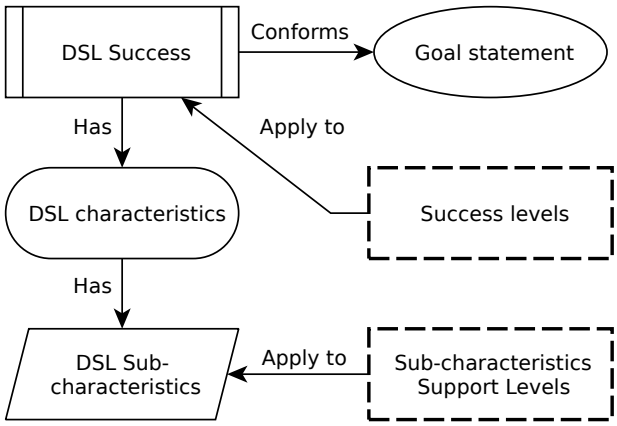


Figure 6.1: *FQAD Assessment Model Components. Adapted from [27]*

The DSL success assessment process consists of three steps: first, the evaluator assigns an importance ranking to each of the selected DSL quality characteristics. This is done according to the alignment of the characteristic with the assessment goal. Second, based on the feedback of the language provided by users participating in the evaluation process, the evaluator determines the support level for each characteristic. Finally, the results of the assessment are obtained according to the rules defined in the FQAD assessment model.

In order to asses the effectiveness of Pascani and Amelia, we designed a set of exercises to use each language in a controlled development environment, and a questionnaire for assessing the experience. We apply the whole exercise with a set of evaluators in evaluation sessions called workshops. These exercises are based on the Matrix-Chain Multiplication problem, a case study

we developed in the context of this thesis, that is presented in the next section. We also used these sessions to evaluate the applicability of the dynamic performance monitoring architecture.

6.1 Case Study: The Matrix-Chain Multiplication Problem

The Matrix-Chain Multiplication (MCM) problem is an optimization problem that consists in finding the most efficient multiplication sequence to multiply a set of given matrices. Our implementation of the MCM, provided to the workshop participants, splits the problem into three different subproblems: the matrix-pair multiplication problem, the matrix-chain parenthesization problem, which finds the optimal sequence of matrix-pair multiplications minimizing the number of individual additions and multiplications, and the matrix-subchain multiplication scheduling problem, which finds subsets of matrix multiplications that can be performed concurrently to decrease the overall multiplication time [30]. In this way, by combining the different solutions to these subproblems, it is possible to configure several different actual solutions to the whole problem, which raises a problem of solution configuration. For instance, by combining the first and second subproblems, one can obtain a solution able to multiply a set of given matrices reducing the number of individual arithmetical operations. In the same sense, by combining the first and third subproblems, the same solution mentioned above would be obtained, but this time reducing multiplication time. And of course, by combining the three subproblems both operations and overall processing time would be reduced. In practice, however, there can be computational limitations and trade-offs that may make infeasible some of the possible solution configurations.

The following figure depicts the variability of the MCM configurations to build a concrete solution using a feature model [13].

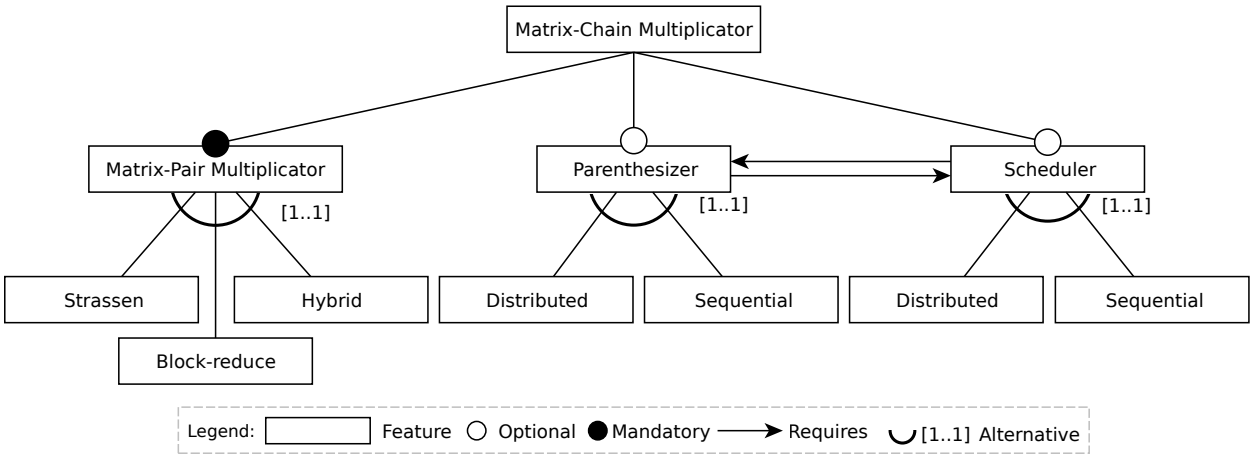


Figure 6.2: *Features Diagram of the MCM configurations*

In this implementation of the MCM solution, we take advantage of distributed computational

resources in order to reduce the execution time when multiplying a large number of considerably big matrices. To this end, we developed two multiplication strategies, one based on the map-reduce architecture, and a variation of it that significantly reduces network usage. At the end, local multiplications are performed using the Strassen algorithm.

The following deployment diagrams depict the high-level elements composing each of the multiplication strategies. For sake of simplicity, we omit the details of the scheduling and parenthesizing subproblems. As there is only one artifact per strategy (*i.e.*, one resulting artifact of the compilation process), a note on each diagram specifies the node in which the components are executed.

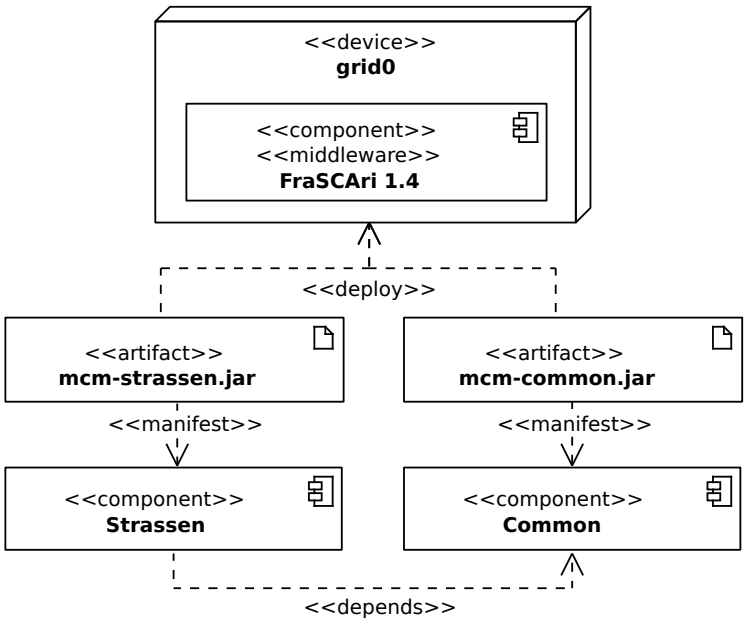


Figure 6.3: *Deployment Diagram for the Monolithic Strassen Configuration Strategy*

The monolithic Strassen configuration strategy considers only a multiplication component that takes the sequence of matrices as it is, and multiplies them iteratively in one computing node. This strategy leaves out the optimizations introduced by subproblems matrix-chain parenthesization and matrix-subchain multiplication scheduling.

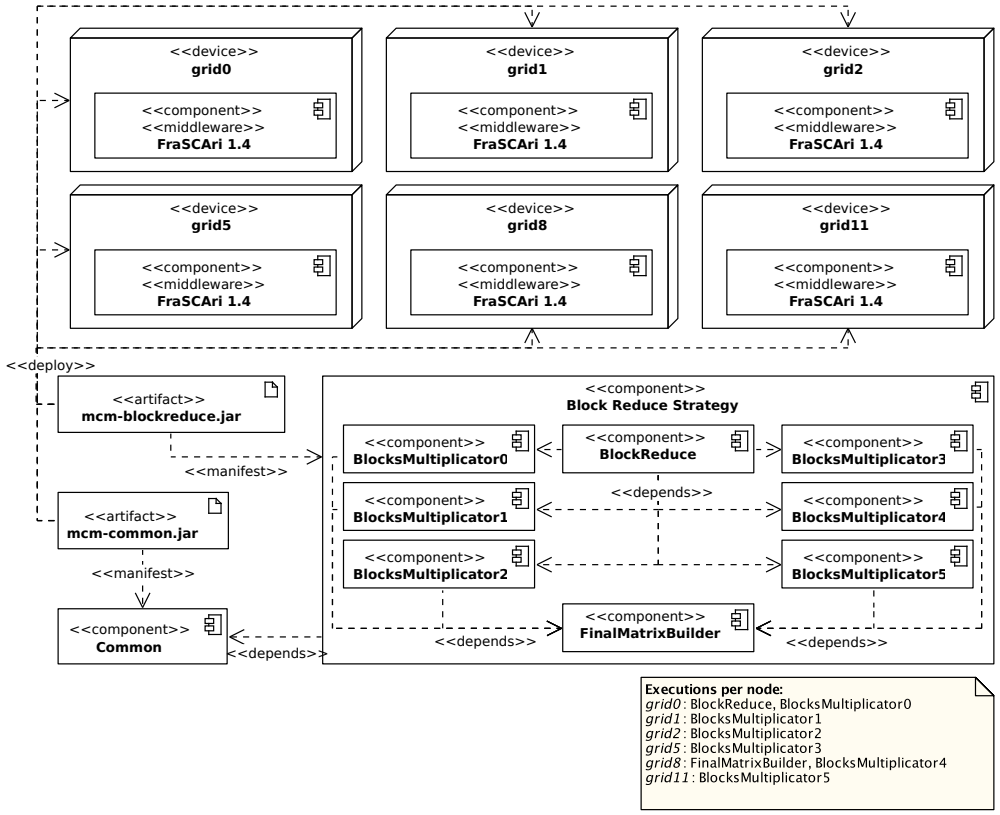


Figure 6.4: *Deployment Diagram for the BlockReduce Configuration Strategy*

The BlockReduce consiguration strategy consists in splitting each matrix into fixed-size blocks (*i.e.*, sub-matrices) and multiply them as if they were one cell instead of a group of them. For instance, having two squared matrices A and B, partitioned into 4 blocks each, the resulting matrix C would be calculated using the same blocks partition strategy. C_{00} represents the first block of C, and would be calculated by operating A and B such that $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$. In this strategy, determining the block size is crucial to balance the amount of data transmitted over the network and the size of the blocks to multiply, in order to reduce the multiplication time. We performed several experiments and found that for matrices of approximately 3600x3600 elements, the block size with best execution times is 200.

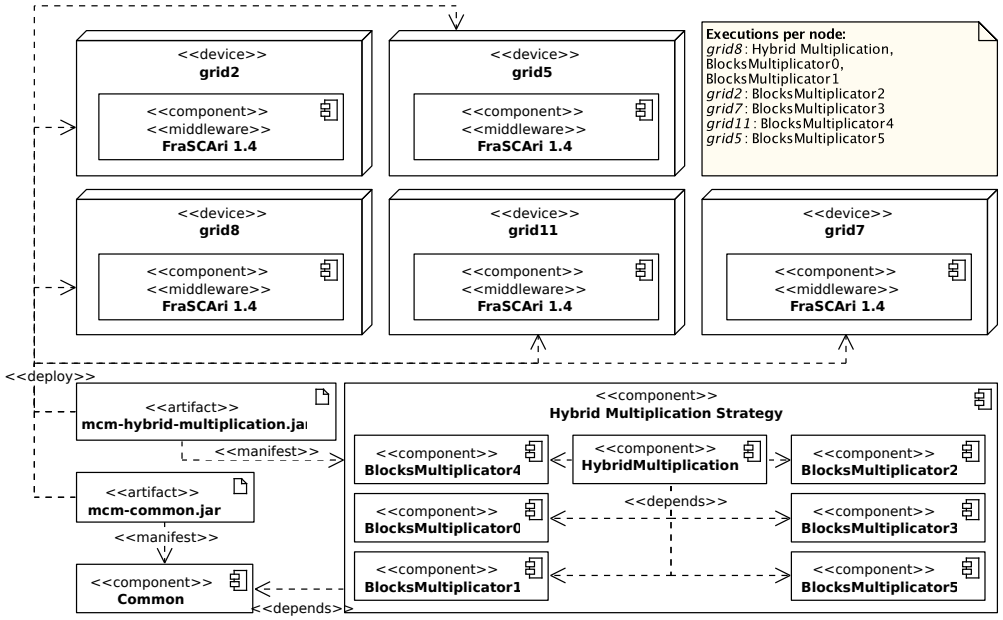


Figure 6.5: *Deployment Diagram for the Hybrid Configuration Strategy*

The Hybrid configuration strategy introduces an improvement in terms of network usage, with respect to the BlockReduce configuration strategy. However, it is more demanding in terms of processor and memory usage. In the strategy above, calculating a block in the resulting matrix requires sending as many pairs of blocks as columns or rows of blocks are, while in this strategy it only requires sending the whole column and row of blocks. Another advantage of this strategy is that it also reduces the amount of processors necessary to multiply the blocks.

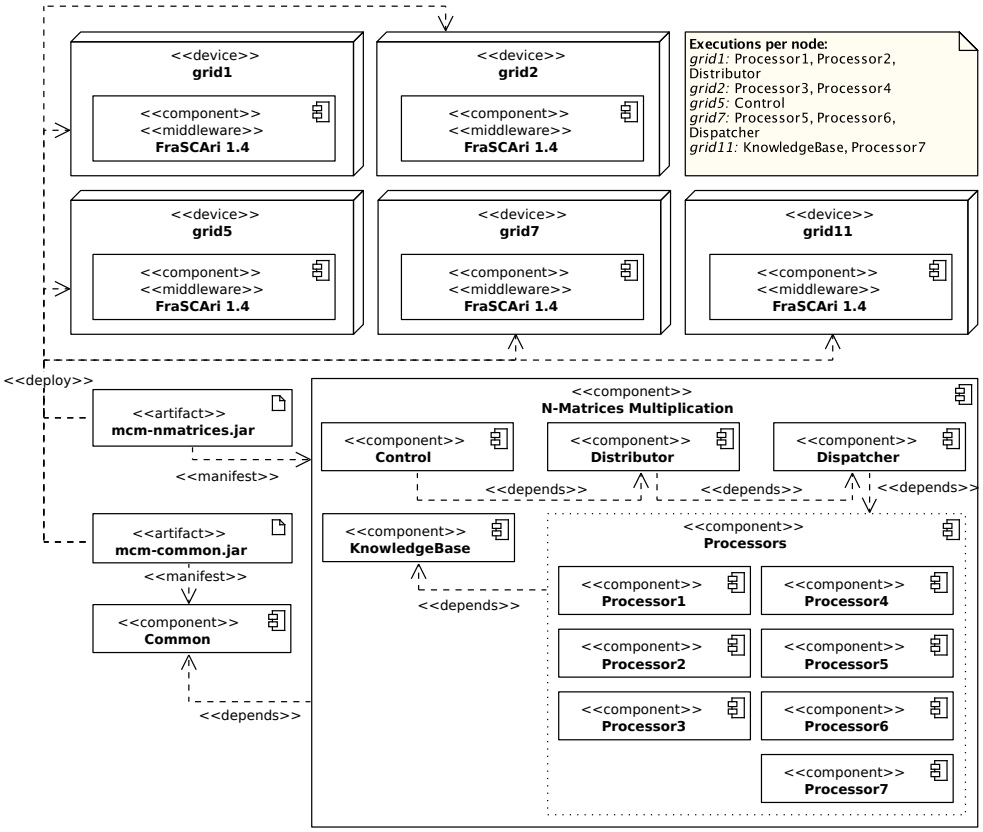


Figure 6.6: *Deployment Diagram for the N-Matrices Configuration Strategy*

6.2 Application of the Assessment Model

Before starting the assessment process, we must select the quality characteristics in order to define the Pascani and Amelia success factors. Given our initial set of requirements, the intended use for each language, and the scope of this thesis, we do not consider as goal the following characteristics: Maintainability, Extendability, Compatibility, and Integrability. This does not necessarily mean that these characteristics have been completely ignored in the design of Pascani and Amelia, instead, it means that for this evaluation they are not considered as key success factors. Therefore, those characteristics are not shown in the results.

6.2.1 **Step 1**: Assignment of Importance Degree

Each DSL quality characteristic is given an importance degree to determine the expectations of an evaluator from a DSL. Importance degrees are defined using an ordinal scale with the following scale levels: mandatory, desirable, and nice to have. Table 6.1 depicts the mapping between the importance degrees defined in FQAD and their required support level. An importance degree can be accepted as fulfilled if the sub-characteristic support level matches with the one below in the

aforementioned table.

Table 6.1: *Mapping between Importance degree and its required Support level*

Importance Degree	Support Level	Support Level Description
Nice to have	No support	Fails to recognize the sub-characteristic. The sub-characteristic is not supported nor referred to in the DSL.
Desirable	Some support	The sub-characteristic is supported but not satisfactorily. It needs improvement.
Mandatory	Strong support	The DSL meets the sub-characteristic.
–	Full support	All aspects of the sub-characteristic are covered and the DSL provides beyond the sub-characteristic requirements.

We define the importance degrees for the PASCANI and AMELIA quality characteristics as follows: Functional suitability, Reliability, and Productivity are *desirable*, whereas Usability and Expressiveness are *mandatory*.

6.2.2 **Step 2**: Determination of Sub-characteristics Support Level

Each workshop questionnaire contains a set of statements regarding the sub-characteristics support in the language. Workshop participants are asked in what extent they agree or disagree with each statement, using the following cardinal scales: strongly agree, agree, neutral, disagree, and strongly disagree. We leave out responses where the participant neither agree nor disagree with the statement (*i.e.*, neutral responses). In these cases, we consider that the participant either did not understand the question, or lacks experience to evaluate the sub-characteristic under assessment. Table 6.2 shows the mapping between our evaluation questionnaire cardinal scales and sub-characteristic support levels.

Table 6.2: *Mapping between Questionnaire cardinal scales and Sub-characteristic support Levels*

Questionnaire Cardinal Scale	Support Level
Strongly agree	Full support
Agree	Strong support
Disagree	Some support
Strongly disagree	No support

Table 6.3 shows the PASCANI and AMELIA support level per quality sub-characteristic, according to the responses of the workshop participants. Each sub-characteristic was given the support level on which more developers agreed, that is, the one that appeared more in the questionnaire answers.

In case of a tie, the lowest support level was selected.

Table 6.3: *Sub-characteristics' Support Level (SL) in* Pascani *&* Amelia*;*

Quality Characteristic	Sub-characteristic	Pascani SL	Amelia SL
Functional Suitability	Completeness	Some Support	Full support
	Appropriateness	Strong Support	Full support
Usability	Comprehensibility	Full support	Strong support
	Learnability	Strong support	Strong support
	Likeability	Strong support	Full support
	User perception	Strong support	Full support
	Operability	Strong support	Strong support
	Compactness	Strong support	Full support
Reliability	Model checking	Strong support	Strong support
	Correctness	Strong support	Full support
Productivity	Development time	Full support	Full support
	Amount of human resource	Strong support	Full support
Expressiveness	Mind to program mapping	Strong support	Full support
	Uniqueness	Strong support	Strong support
	Orthogonality	Strong support	Strong support
	Correspondence to important domain concepts	Strong support	Strong support
	Conflicting elements	Full support	Full support
	Right abstraction level	Strong support	Full support

6.2.3 **Step 3**: Success Level Determination

Success levels dictaminate an overall DSL evaluation in terms of its quality achievement. According to FQAD, for a DSL to fulfill a specific success level, it must satisfy all of the sub-characteristics of the characteristics. FQAD defines the rules of success level determination as follows:

Incomplete DSL is incomplete in satisfying its intended purpose and it needs improvements.

Satisfactory DSL satisfies its intended purpose on average, yet it can be further improved.

Effective DSL satisfies its intended purpose.

Having into account the sub-characteristic support levels presented in Step 2, we state the success level of both Pascani and Amelia as *effective*.

Throughout the realization of the workshop exercises, the language users showed a significant improvement in the specification times, as well as the time spent correcting errors during execution trials. In the case of Amelia specifications, all the exercises improved at least 90% in deployment time when compared to manual deployments. Regarding Pascani, we cannot calculate the improvement in development times given that we have no previous information about manual developments. Furthermore, a fair comparison of development times should consider two applications fulfilling the same set of requirements, which makes it more difficult to accomplish. This improvement in developing specifications reflects the effectiveness of both languages in the evaluated quality characteristics.

Chapter 7

Conclusions and Future Work

Continuous service delivery and accomplishment of agreed levels of fulfillment in service performance requires insightful information on the current system state, regarding not only the hardware infrastructure but also its constituting software components. Moreover, advances in autonomic computing for strengthening service responsiveness and resilience have promoted the design of re-configurable systems able to modify its structure and behavior at runtime. Then, to actually ensure the continuous satisfaction of performance factors in these systems, monitoring infrastructures must be able of (i) dynamically updating its monitoring strategies as the system's requirements or the environment evolves; and (ii) realizing the deployment and integration of monitoring components at runtime. Moreover, in providing the system itself with self-awareness mechanisms (*i.e.*, mechanisms enabling the system with awareness regarding its own behavior), such infrastructure must also (iii) provide the means to generate composable, traceable, and controllable monitoring capabilities.

In order to provide a solution satisfying the challenges presented above, we analyze and break down the stated needs into functional requirements and quality considerations. These requirements are first classified by component, in Monitors and Probes, and then by stage of the monitoring process in Data acquisition, Data aggregation and filtering, Data persistence, and Data visualization. The quality concerns identified are related to the dynamic deployment and re-deployment of the infrastructure's elements, their composability and controllability, and the scalability of the infrastructure.

This thesis propose a component-based dynamic monitoring architecture to overcome the challenges identified, and meet the stated requirements. In order to abstract operative and low-level technical details, we created Pascani and Amelia, two domain-specific languages for generating the envisioned monitoring components, and deploying and re-deploying them into the running infrastructure, respectively. Our solution constitutes an effort to advance in the development of self-

awareness mechanisms, and furthermore, in the realization of DYNAMICO's monitoring feedback loop.

7.1 Technical Limitations

In the development of our solution, we have encountered technological limitations that did not allow us to provide certain functionalities such as measuring network communication times with Pascani, and binding RMI services at runtime with Amelia. Moreover, we have made some decisions about the scope of our proof-of-concept implementation that have left out some functional requirements for future development, such as state recovery after re-deployment, and configuration of development environments with Pascani and Amelia, respectively. This section provides the details of a selected list of these limitations.

Configuration of Development Environments

The Amelia language has been designed to deploy distributed component-based systems. Deploying such systems includes activities such as source code compilation, execution and binding configuration. However, deployment also involves the activities related to the configuration of execution environments, such as installing an operating system, configuring the hardware properties of the machines, configuring a firewall, among many other activities. Advances in management of cloud infrastructures have promoted the virtualization of many of these tasks, and today there are many tools and languages specialized in the creation and configuration of development/production environments. Amelia does not directly support these type of tasks, however, as the Amelia runtime is able to comunicate with remote computing nodes using SSH sessions, it could certainly execute the necessary commands to trigger a desired set of environment configuration tasks.

Network Communication Probes

All the runtime interception operations in the dynamic monitoring infrastructure are supported by the FraSCAti middleware. Although FraSCAti has been designed for introspecting SCA applications, measuring network communication time requires not only to intercept one component, but two sides of communication, within the context of a single service invocation. For measuring the time it takes to send data over the network, one have to know the invocation's start and end time; the problem remains in that those times are measured at different locations, at the service consumer and service provider components. The problem here is that invocations are not identifiable, they do not have a unique key to which both times can be attached. This is a common issue when measuring network communication times. A good approach used by RPC servers is the *stubs and skeletons* strategy. This strategy considers the standard generation of two components, an stub and an skeleton, for sending a transaction ID and the invocation start time, along with the original request from the stub to the skeleton. Then, the skeleton removes the additional information, and

forwards the request to the service provider. given this thesis' scope, the current version of Pascani lacks of an implementation of the network communication probe.

Service Binding at Runtime

Since Amelia was implemented to deploy SCA components generated by Pascani, FraSCAti is the Amelia's target middleware platform. This limits Amelia in the kind of bindings that can be performed at runtime to web and REST services, leaving out RMI bindings.

State Recovering on Re-deployment

The current implementation of Pascani does not includes a mechanism to recover the state after a re-deployment is performed. Since monitor elements are event-driven, this does not entail serious consequences in terms of losing the current state. For namespaces, however, this is a relevant issue, as monitors would continue working with the default values of the context variables, instead of the ones that reflect the system state. Recovering a namespace's state after a re-deployment requires to initialize its variables from the values stored in the database; this is a complex process, given that there must exist an standard mapping mechanism to marshall and unmarshall any data type. Implementing such a mechanism would have to solve many of the problems of object-relational mapping (ORM) technologies. Pascani currently supports only the storage of primitive data types (*i.e.*, the marshall process), but with little effort the unmarshalling for such data types can be supported.

7.2 Future Work

Evolution of Pascani and Amelia

According to our evaluation, both Pascani and Amelia provide an appropriate level of domain abstraction, and improve development productivity. However, we still need to perform tests regarding performance monitoring and system deployment in different types of software applications. Although we designed both languages concerned with comprising as much concepts as possible from each language domain, different kinds of requirements may arise. Consequently, we may need to adapt and evolve the syntax and semantics of each language.

Development of Self-awareness Mechanisms

The architecture we propose in this thesis allows the manual specification of monitoring specifications, and the automated generation and deployment of monitoring components. In order to move the state of the art forward, and reach always-relevant self-awareness mechanisms, we need to augment our solution with autonomic capabilities to continuously evolve the monitoring infrastructure.

One way to this is by designing and implementing the DYNAMICO's monitoring feedback loop based on our architecture for dynamic performance monitoring.

Towards Supporting Automatic Generation of Pascani and Amelia Specifications

One of our ultimate objectives in developing dynamic monitoring mechanisms is to provide a base line to, gradually, enable the system itself for generating the monitoring components that allow the infrastructure to remain pertinent. However, the syntax and semantics of Pascani are still in a low-level abstraction that makes it difficult for the system to assemble new specifications in order to fulfill emerging monitoring requirements. We consider that an adequate solution to this problem is to abstract Pascani specifications into high-level policies. These policies would considerably minimize monitoring specifications, while hiding irrelevant technical details for the system. For instance, to observe latency in a given service, the system would have to declare an event with the corresponding target service, create an event handler with the logic to update the latency variable, and a configuration block for subscribing the handler to the execution event. Additionally the system would have to declare the monitor's package and name, and import the required Java classes. This becomes complex when there are several context variables and services involved in the monitoring requirement, and even more complex when monitoring requirements are dependent among them. Such cases would probably require to design a derivation strategy based on composable templates and fragments. Nonetheless, abstracting those monitoring requirements into policies with clear scope would make it easier to derive Pascani specifications, and also to represent monitoring requirements/strategies in knowledge sources (*i.e.*, Knowledge element of the MAPE-K reference model).

These considerations also apply to the Amelia language. Deploying a component requires the declaration of on-host expressions, execution rules, and commands. In this case policies would help to reduce the amount of code needed to express a requirement, however, that might not be enough. The deployment of a single component expects to be given the source code directory, artifact name, dependencies, libraries, and commands to compile, possibly configure, and execute it. We consider that the convention-over-configuration principle is a good complement to leverage the system for understanding intricate deployment strategies in a simpler way.

Part I

Appendices

Appendix A

Workshop for Evaluating the Effectiveness of Pascani

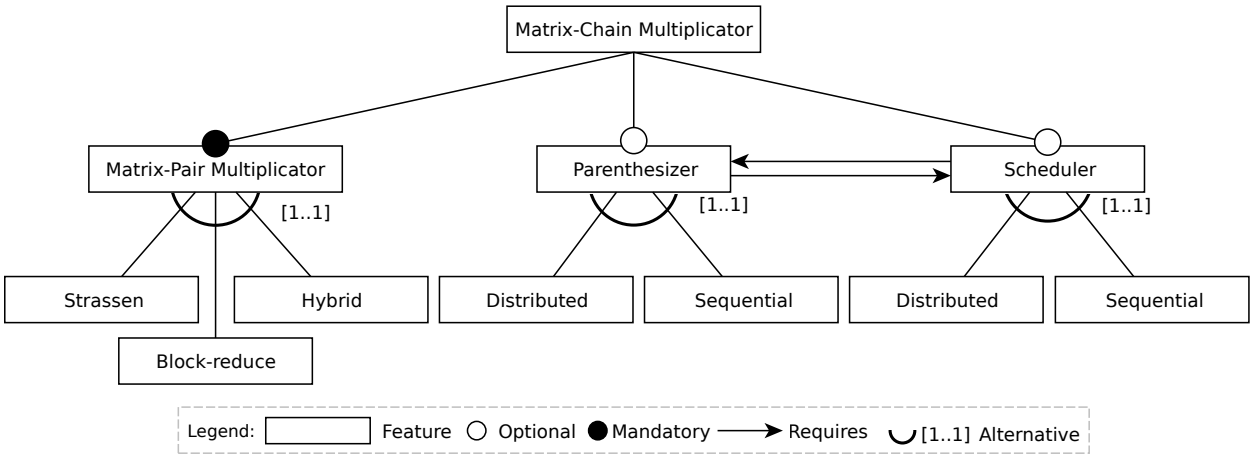
To evaluate the effectiveness of the Pascani DSL, our language that assists application developers in the development of dynamic performance monitors for component-based software systems, we designed a set of exercises to use Pascani, and a questionnaire for assessing the experience of using the language. We call this whole exercise a workshop. The idea for the workshop is to be applied by a group of developers, who will evaluate the effectiveness of the language. In this section, we explain the workshop evaluation design, including the case study of application, a short introduction of the language, and then the set of exercises through which the evaluation process is performed.

A.1 Case Study: The Matrix-Chain Multiplication Problem

The Matrix-Chain Multiplication (MCM) problem is an optimization problem that consists in finding the most efficient multiplication sequence to multiply a set of given matrices. Our implementation of the MCM, provided to the workshop participants, splits the problem into three different subproblems: the matrix-pair multiplication problem, the matrix-chain parenthesization problem, which finds the optimal sequence of matrix-pair multiplications minimizing the number of individual additions and multiplications, and the matrix-subchain multiplication scheduling problem, which finds subsets of matrix multiplications that can be performed concurrently to decrease the overall multiplication time [30]. In this way, by combining the different solutions to these subproblems, it is possible to configure several different actual solutions to the whole problem, which raises a problem of solution configuration. For instance, by combining the first and second subproblems, one can obtain a solution able to multiply a set of given matrices reducing the number of individual arithmetical operations. In the same sense, by combining the first and third subproblems, one would obtain the same solution aforementioned, but this time

reducing multiplication time. And of course, by combining the three subproblems one would reduce both operations and overall processing time. In practice, however, there can be computational limitations and trade-offs that may make infeasible some of the possible solution configurations.

The following figure, in UML notation, depicts the variability of the MCM configurations to build a concrete solution using a feature model [13].



In this implementation of the MCM solution, we take advantage of distributed computational resources in order to reduce the execution time when multiplying a large number of considerably big matrices. To this end, we developed two multiplication strategies, one based on the map-reduce architecture, and a variation of it that significantly reduces network usage. At the end, local multiplications are performed using the Strassen algorithm.

The following deployment diagrams depict the high-level elements composing each of the multiplication strategies. For sake of simplicity, we omit the details of the scheduling and parenthesizing subproblems. As there is only one artifact per strategy (*i.e.*, one resulting artifact of the compilation process), a note on each diagram specifies the node in which the components are executed.

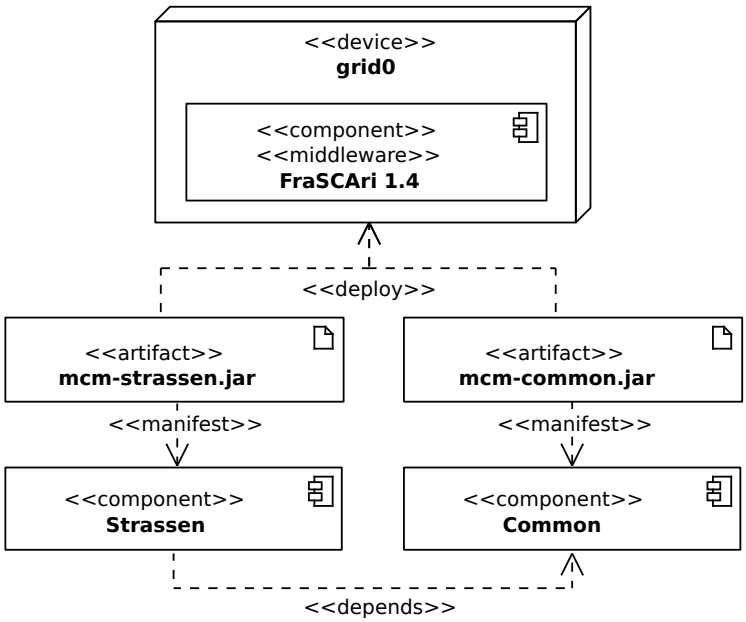


Figure A.1: *Deployment Diagram for the Monolithic Strassen Configuration Strategy*

The monolithic Strassen configuration strategy considers only a multiplication component that takes the sequence of matrices as it is, and multiplies them iteratively in one computing node. This strategy leaves out the optimizations introduced by subproblems matrix-chain parenthesization and matrix-subchain multiplication scheduling.

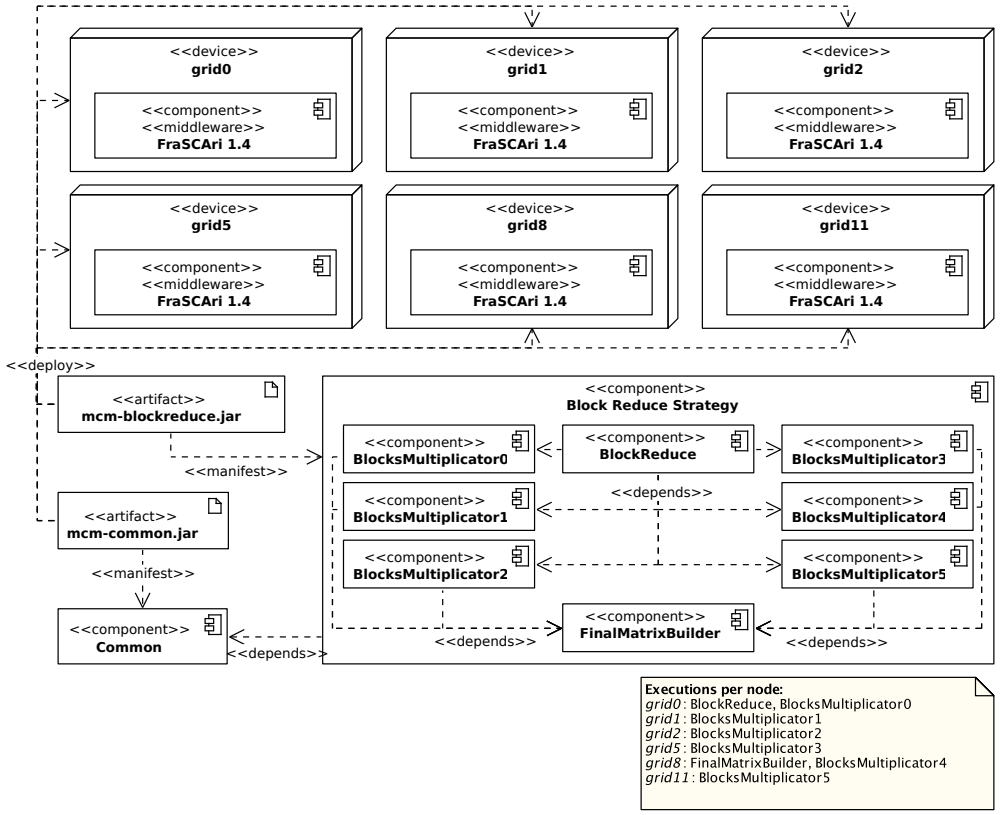


Figure A.2: *Deployment Diagram for the BlockReduce Configuration Strategy*

The BlockReduce consiguration strategy consists in splitting each matrix into fixed-size blocks (*i.e.*, sub-matrices) and multiply them as if they were one cell instead of a group of them. For instance, having two squared matrices A and B, partitioned into 4 blocks each, the resulting matrix C would be calculated using the same blocks partition strategy. C_{00} represents the first block of C, and would be calculated by operating A and B such that $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$. In this strategy, the block size is crucial to find the threshold between the amount of data transmitted over the network and the size of the blocks to multiply, in order to reduce the multiplication time. We performed several experiments and found that for matrices of approximately 3600x3600 elements, the block size with best execution times is 200.

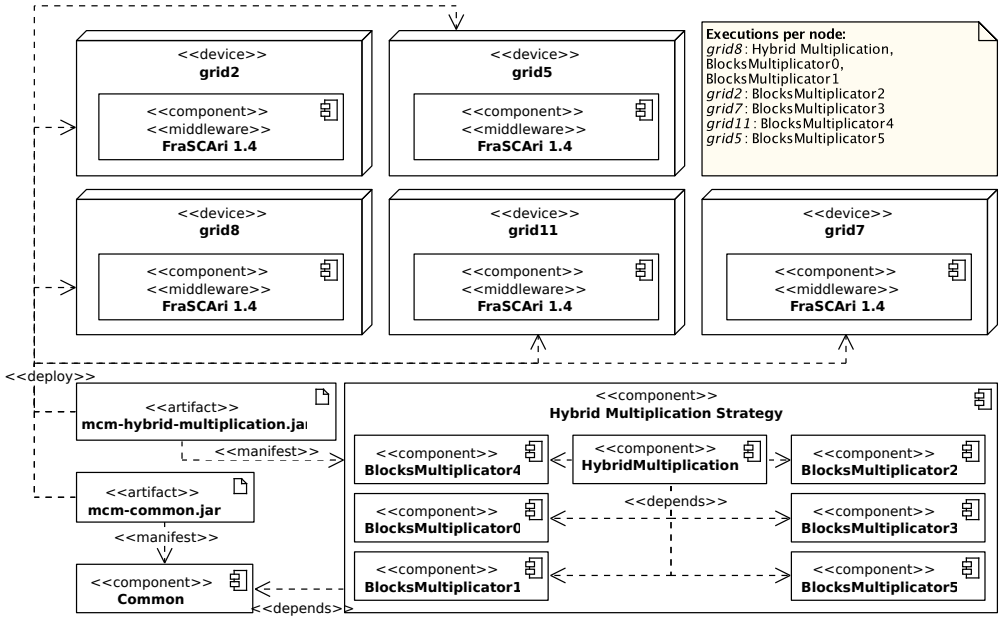


Figure A.3: *Deployment Diagram for the Hybrid Configuration Strategy*

The Hybrid configuration strategy introduces an improvement, in terms of network usage, to the BlockReduce configuration strategy. However, it is more demanding in terms of processor and memory usage. In the strategy above, calculating a block in the resulting matrix requires sending as many pairs of blocks as columns or rows of blocks are, while in this strategy it only requires sending the whole column and row of blocks. Another advantage of this strategy is that it also reduces the amount of processors necessary to multiply the blocks.

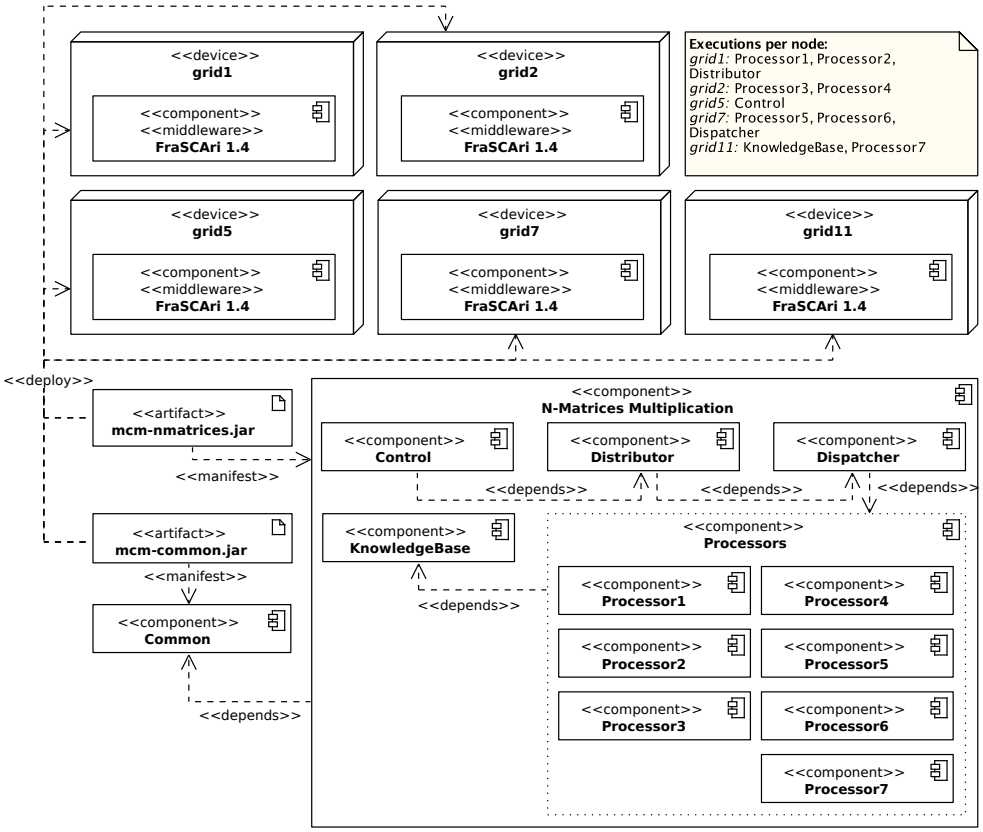


Figure A.4: *Deployment Diagram for the N-Matrices Configuration Strategy*

A.2 The PASCANI Language: Specification Examples

PASCANI allows specifying two types of constructs: Namespaces and Monitors. A namespace is hierarchical structure that allows defining, reading and updating context variables. A monitor is an entity to continuously observe and check the state of the system or a particular subsystem regarding a set of performance factors. Along this workshop you will be given an overview of the language syntax and its associated semantics.

Listings A.1 and A.2 are namespaces that will be used in the specification example, and in the practical exercises as well. The first one contains the reference values for the performance level indicators; and the second one contains the context variables representing the actual system state.

```
1 package co.edu.icesi.driso.matrices
2 import org.quartz.CronExpression
3 /*
4  * Service Level Indicators & other reference variables
5  */
```

```
namespace SLI {
   /*
    * Expected throughput in a period of 10 seconds
    */
   val Integer throughput = 10

   /*
    * Chronological expression representing the throughput period
    */
   val CronExpression throughputPeriod = '*/10 * * * * ?'

   /*
    * Expected latency for all service executions
    */
   val Integer latency = 3000
}
```

Listing A.1: *Reference values*

```
package co.edu.icesi.driso.matrices
/*
 * Monitoring variables representing the actual system state
 */
namespace State {
   /*
    * Represents the number of multiplications done in the
    * latest throughput period
    */
   var Integer throughput = 0

   /*
    * Represents the service latency (for all executions)
    */
   var Long latency = 0L
}
```

Listing A.2: *Context variables*

The following examples describe two monitoring requirements and their corresponding solutions in Pascani.

A.2.1 Requirement 1 (Example)

Measure the execution time in all invocations performed to the multiplication service of the Strassen component strategy, and update the `latency` variable adding the contextual information *strategy*, *host*, and *component*.

In PASCANI this would require to declare an *invocation* or *return* event whose target is the Strassen component, and every time the event is triggered, the variable is updated. The type of event to declare here is important because if an exception is thrown while a request is being served, the execution time should not be measured.

```
1  package co.edu.icesi.driso.matrices.strassen
2
3  import java.net.URI
4  import org.pascani.dsl.lib.events.ReturnEvent
5  import static org.pascani.dsl.lib.sca.FluentFPath.$domain
6
7  using co.edu.icesi.driso.matrices.State
8
9  monitor Latency {
10    val target = $domain.child("Strassen").child("matrix").service("multiplication")
11
12    event e raised on return of target
13
14    handler onReturn(ReturnEvent e) {
15       val tags = #{
16          "strategy" -> "strassen",
17          "host" -> "grid0",
18          "component" -> "Strassen"
19       }
20       State.latency = tag(e.value, tags)
21    }
22
23    config {
24       e.bindingUri = new URI("http://grid0:" + 3000)
25       e.subscribe(onReturn)
26    }
27 }
```

Line 7 declares that monitor `Latency` reads and updates the variables declared within the namespace `State`. Line 10 declares an immutable value holding an FScript expression specifying the `multiplication` service provided by component `Strassen`. Line 12 declares a *return* event on the

multiplication service. In line 20, the latency variable is updated and tagged with contextual information. And lines 24 and 25 configure the FraSCAti reconfiguration URI and subscribe the `onReturn` handler to the return event `e`, respectively.

A.2.2 Requirement 2 (Example)

Measure the throughput of the Strassen strategy and update the `throughput` variable adding the contextual information *strategy*, *host*, and *component*.

In Pascani this would require to introduce a probe into the Strassen component, and periodically count the number of served requests (*i.e.*, the number of successful invocations).

```
1  package co.edu.icesi.driso.matrices.strassen
2
3  import java.net.URI
4  import org.pascani.dsl.lib.events.IntervalEvent
5  import org.pascani.dsl.lib.events.ReturnEvent
6  import static org.pascani.dsl.lib.sca.FluentFPath.$domain
7
8  using co.edu.icesi.driso.matrices.SLI
9  using co.edu.icesi.driso.matrices.State
10
11 monitor Throughput {
12    val target = $domain.child("Strassen").child("matrix").service("multiplication")
13    val routingKey = "strassen.throughput"
14    val bindingUri = new URI("http://grid0:" + 3000)
15    val probe = newProbe(target, routingKey, ReturnEvent, false, bindingUri)
16
17    event i raised periodically on SLI.throughputPeriod
18
19    handler onInterval(IntervalEvent e) {
20       val count = probe.countAndClean(-1, System.currentTimeMillis)
21       val tags = #{
22          "strategy" -> "strassen",
23          "host" -> "grid0",
24          "component" -> "Strassen"
25       }
26       State.throughput = tag(count, tags)
27    }
28
29    config {
30       i.subscribe(onInterval)
```

```
31     }
32 }
```

Lines 8 and 9 declare that monitor `Throughput` reads and updates the variables declared within namespaces `SLI` and `State`. Line 15 introduces a new monitor probe into the `multiplication` service for collecting *return* events. Line 17 declares a time-based event using the throughput period defined in SLI. In line 26, the throughput variable is updated and tagged with contextual information. And line 30 subscribes the `onInterval` handler to the periodic event `i`.

A.3 Practical Exercises

Use PASCANI to specify the monitors solving the following monitoring requirements. In all exercises, be sure of updating the corresponding context variables with the contextual information *strategy*, *host* and *component*. Please measure the time you spend while solving each exercise. After finishing the exercises, you will be given a questionnaire to evaluate a set of success factors regarding PASCANI.

Exercise 1: Measure the latency for the Block-reduce strategy, registering only the overall latency.

Exercise 2: Measure the latency for the Block-reduce strategy, registering the corresponding values for all of the components involved in the process of calculation.

Exercise 3: Measure the throughput for the Hybrid-multiplication strategy, registering only the overall throughput.

Exercise 4: Send an email when the throughput value is less than the specified in the SLI namespace. For solving this exercise, you will be given an interface and the binding information of an already deployed email service.

Exercise 5: Define two more monitoring requirements for the case of study and explain how you would solve them using PASCANI.

A.4 Questionnaire for Evaluating PASCANI

1. For each exercise, please indicate the time you spent (in minutes) specifying it.
E1. _____ E2. _____ E3. _____ E4. _____ E5. _____

2. Please describe the difficulties you experienced developing this workshop.

_____

_____

_____

3. How much time have you been working as a professional Software Engineer? _____

4. Please describe how much experience you have developing software tests or monitors.

_____

_____

_____

5. Please describe how much experience you have with PASCANI.

_____

_____

_____

Please indicate your agreement or disagreement with the following statements by selecting only one square. The left-most square indicates that you **strongly agree**, while the right-most square indicates that you **strongly disagree**.

Functional Suitability

6. All concepts and building-blocks for solving problems in the software monitoring domain can be expressed in PASCANI. □—□—□—□—□

7. PASCANI is an appropriate and useful tool for specifying software monitors. □—□—□—□—□

Usability

8. The language elements are understandable (*e.g.*, language elements can be understood after reading their descriptions). □—□—□—□—□

9. The concepts and symbols of the language resemble the terminology of the monitoring domain, are learnable and rememberable (*i.e.*, learning easiness, easiness for developing monitor specifications). □—□—□—□—□

10. PASCANI helps users achieve their tasks in acceptable development times. □—□—□—□—□

11. PASCANI is appropriate for developing the type of performance monitors you need. □—□—□—□—□

12. PASCANI has useful language elements to operate on monitor data and control the actual monitoring operations (*e.g.*, language elements can be selected and put into practice easily, actions are undoable, error messages that explain recovery methods are available for controlling the monitoring operations). □—□—□—□—□

13. PASCANI has a concise syntax that allows expressing performance monitors in short specification files. □—□—□—□—□

Reliability

14. PASCANI prevents making errors in monitoring specifications. The language constructs helps the user to avoid mistakes. □—□—□—□—□

15. PASCANI includes the right elements and correct relationships between them (it prevents unexpected interactions between its elements). □—□—□—□—□

Maintainability

16. PASCANI is composed of discrete components such that a change to one component has minimal impact on other components. □—□—□—□—□

Productivity

17. The development time of writing software monitors specifications is improved. □—□—□—□—□

18. PASCANI helps to improve the productivity of monitor development. □—□—□—□—□

Expressiveness

19. A monitoring strategy can be mapped into a PASCANI specification easily. □—□—□—□—□

20. PASCANI provides one and only one good way to express every concept of interest. □—□—□—□—□

21. Each PASCANI construct is used to represent exactly one distinct concept in the application domain. □—□—□—□—□

22. The language constructs correspond to significant application domain concepts. PASCANI does not include domain concepts that are not important. □—□—□—□—□

23. PASCANI does not contain conflicting or ambiguous elements. □—□—□—□—□

24. PASCANI is at the right abstraction level for writing monitoring specifications, such that it is not more complex or more detailed than necessary. □—□—□—□—□

Integrability

25. PASCANI can be integrated with other languages used in the software development process, such as using already developed libraries. (*e.g.*, language integrability with other languages). □—□—□—□—□

Appendix B

Workshop for Evaluating the Effectiveness of Amelia

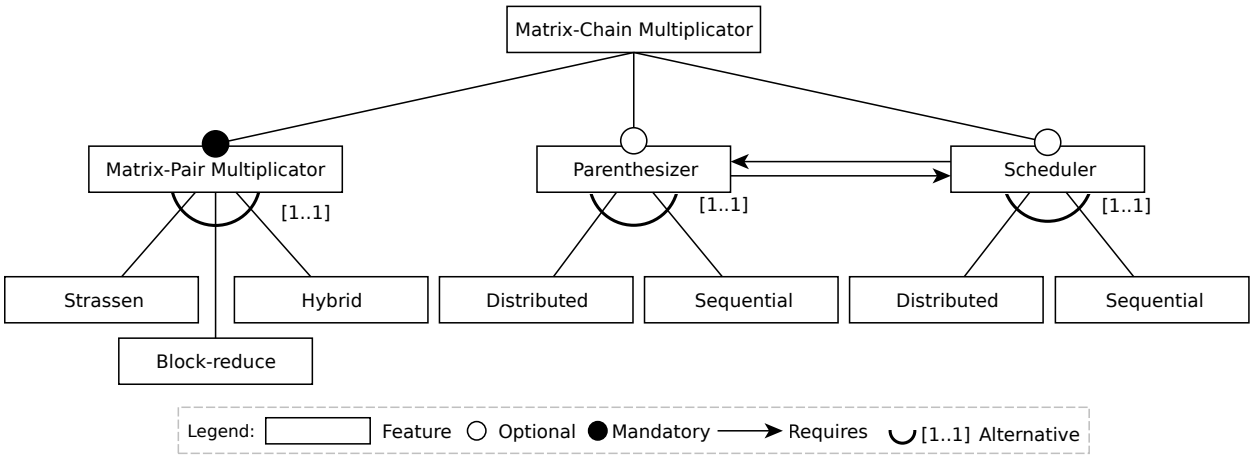
To evaluate the effectiveness of the Amelia DSL, our language that assists application developers in automating the deployment of component-based software systems, we designed a set of exercises to use Amelia, and a questionnaire for evaluating the experience. We call this whole exercise a workshop. The idea for the workshop is to be applied by a group of developers, who will evaluate the effectiveness of the language. In this section, we explain the workshop evaluation design, including the case study of application, a short introduction of the language, and then the set of exercises through which the evaluation process is performed.

B.1 Case Study: The Matrix-Chain Multiplication Problem

The Matrix-Chain Multiplication (MCM) problem is an optimization problem that consists in finding the most efficient multiplication sequence to multiply a set of given matrices. Our implementation of the MCM, provided to the workshop participants, splits the problem into three different subproblems: the matrix-pair multiplication problem, the matrix-chain parenthesization problem, which finds the optimal sequence of matrix-pair multiplications minimizing the number of individual additions and multiplications, and the matrix-subchain multiplication scheduling problem, which finds subsets of matrix multiplications that can be performed concurrently to decrease the overall multiplication time [30]. In this way, by combining the different solutions to these subproblems, it is possible to configure several different actual solutions to the whole problem, which raises a problem of solution configuration. For instance, by combining the first and second subproblems, one can obtain a solution able to multiply a set of given matrices reducing the number of individual arithmetical operations. In the same sense, by combining the first and third subproblems, one would obtain the same solution aforementioned, but this time reducing multiplication time. And of course, by combining the three subproblems one would reduce

both operations and overall processing time. In practice, however, there can be computational limitations and trade-offs that may make infeasible some of the possible solution configurations.

The following figure, in UML notation, depicts the variability of the MCM configurations to build a concrete solution using a feature model [13].



In this implementation of the MCM solution, we take advantage of distributed computational resources in order to reduce the execution time when multiplying a large number of considerably big matrices. To this end, we developed two multiplication strategies, one based on the map-reduce architecture, and a variation of it that significantly reduces network usage. At the end, local multiplications are performed using the Strassen algorithm.

The following deployment diagrams depict the high-level elements composing each of the multiplication strategies. For sake of simplicity, we omit the details of the scheduling and parenthesizing subproblems. As there is only one artifact per strategy (*i.e.*, one resulting artifact of the compilation process), a note on each diagram specifies the node in which the components are executed.

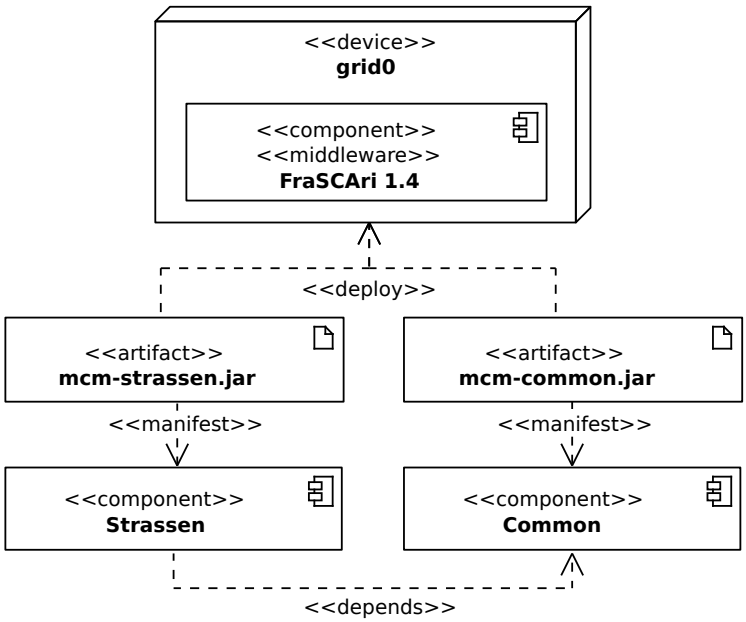


Figure B.1: *Deployment Diagram for the Monolithic Strassen Configuration Strategy*

The monolithic Strassen configuration strategy considers only a multiplication component that takes the sequence of matrices as it is, and multiplies them iteratively in one computing node. This strategy leaves out the optimizations introduced by subproblems matrix-chain parenthesization and matrix-subchain multiplication scheduling.

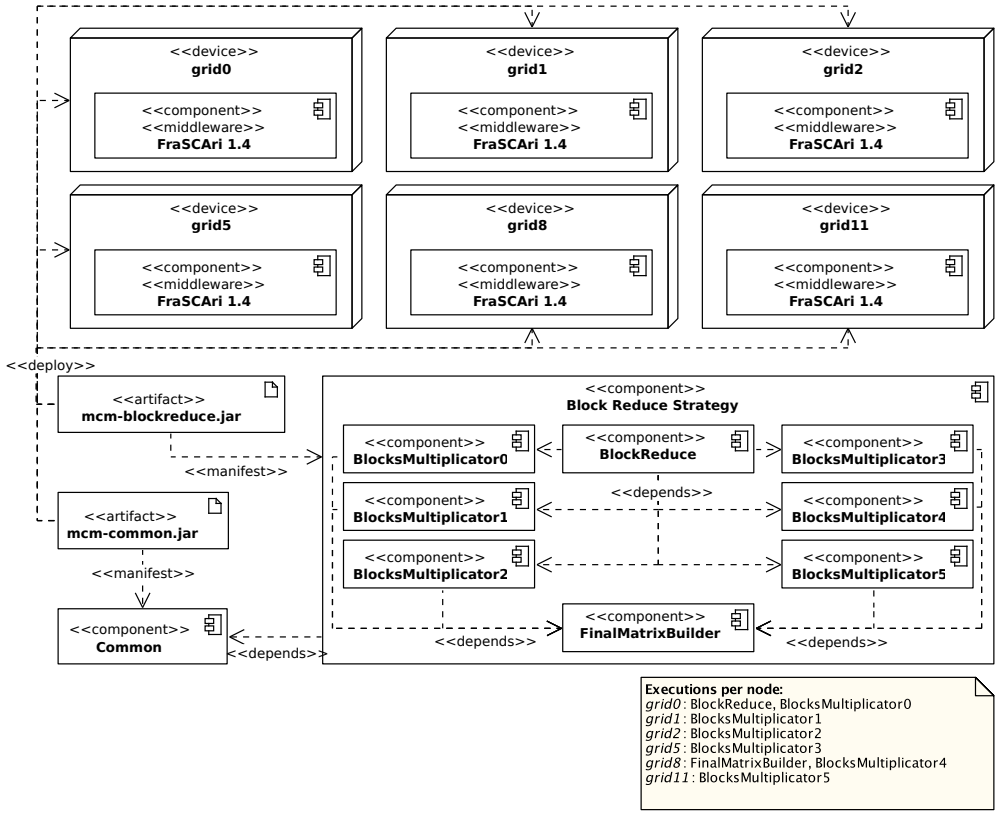


Figure B.2: *Deployment Diagram for the BlockReduce Configuration Strategy*

The BlockReduce consiguration strategy consists in splitting each matrix into fixed-size blocks (*i.e.*, sub-matrices) and multiply them as if they were one cell instead of a group of them. For instance, having two squared matrices A and B, partitioned into 4 blocks each, the resulting matrix C would be calculated using the same blocks partition strategy. C_{00} represents the first block of C, and would be calculated by operating A and B such that $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$. In this strategy, the block size is crucial to find the threshold between the amount of data transmitted over the network and the size of the blocks to multiply, in order to reduce the multiplication time. We performed several experiments and found that for matrices of approximately 3600x3600 elements, the block size with best execution times is 200.

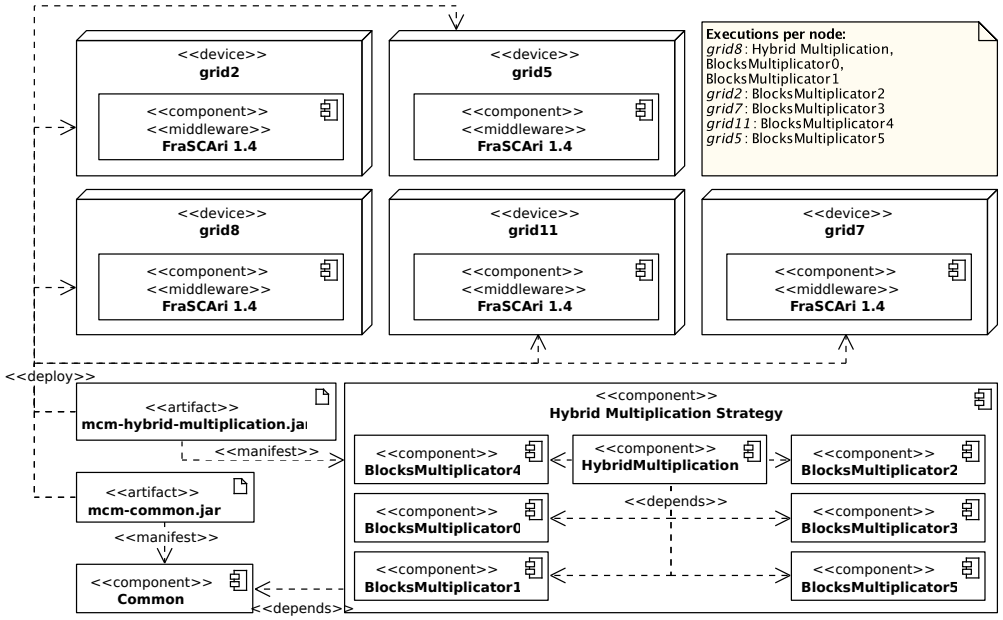


Figure B.3: *Deployment Diagram for the Hybrid Configuration Strategy*

The Hybrid configuration strategy introduces an improvement, in terms of network usage, to the BlockReduce configuration strategy. However, it is more demanding in terms of processor and memory usage. In the strategy above, calculating a block in the resulting matrix requires sending as many pairs of blocks as columns or rows of blocks are, while in this strategy it only requires sending the whole column and row of blocks. Another advantage of this strategy is that it also reduces the amount of processors necessary to multiply the blocks.

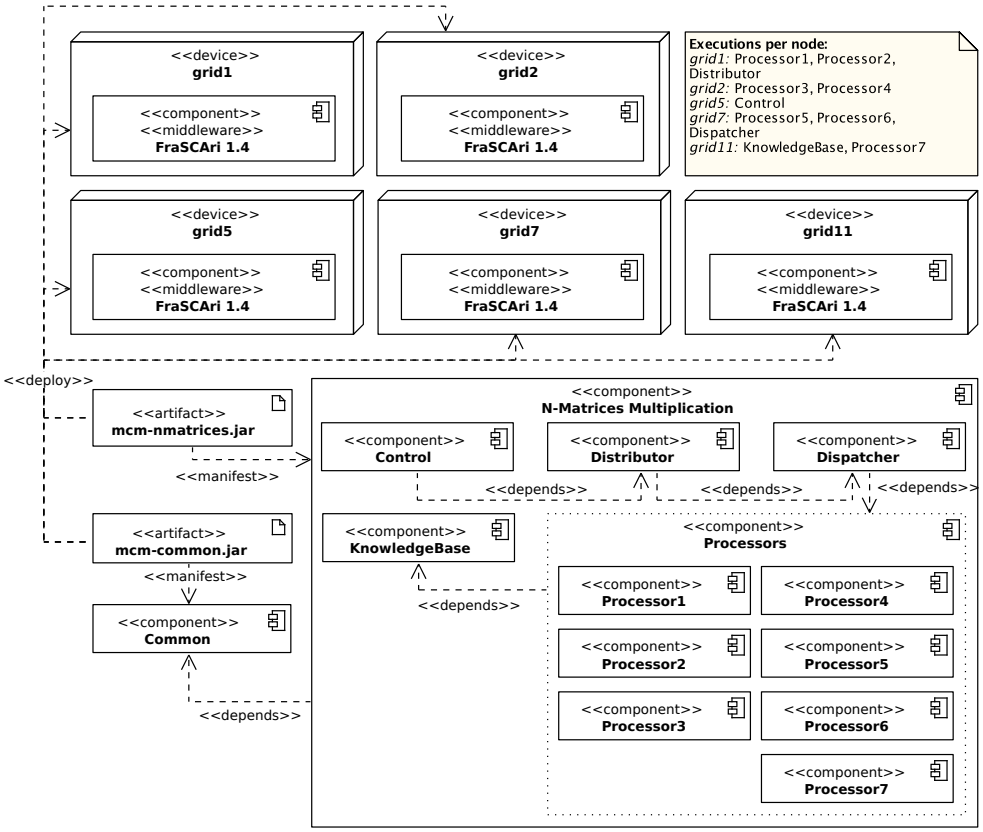


Figure B.4: *Deployment Diagram for the N-Matrices Configuration Strategy*

B.2 The Amelia Language: Specification Examples

Amelia allows specifying two types of elements: Subsystems and Deployments. Along with this workshop you will be given an overview of the language syntax and the associated semantics. The following examples describe two deployment requirements and their corresponding solution in Amelia.

B.2.1 Requirement 1 (Example)

Specify the deployment of the Strassen strategy having into account component dependencies and the computing nodes specified in the deployment diagram. You may use the following file containing a mapping between the hosts in the deployment diagrams and the laboratory's computers. For sake of simplicity assume that every node contains the source code in `/tmp/matrices`, and every strategy (including the `Common` component) is in a subdirectory named: *common*, *strassen*, *nmatrices*, *blockreduce*, and *hybrid*.

```
1 localhost   21 22 user  password localhost
```

```
hgrid17  21 22 user  password grid0
hgrid16  21 22 user  password grid1
hgrid15  21 22 user  password grid2
hgrid14  21 22 user  password grid3
hgrid13  21 22 user  password grid4
hgrid12  21 22 user  password grid5
hgrid11  21 22 user  password grid6
hgrid10  21 22 user  password grid7
hgrid9   21 22 user  password grid8
hgrid8   21 22 user  password grid9
hgrid7   21 22 user  password grid10
hgrid6   21 22 user  password grid11
```

Listing B.1: *hosts.txt*

In AMELIA this would require to specify a `subsystem` for generating the corresponding artifacts from the Strassen and Common components' source code. Once they are generated, the Strassen component would be executed. As there are no specific requirements on the deployment, there is no need to specify a custom deployment strategy.

```
package co.edu.icesi.driso.matrices

import java.util.List
import java.util.Map
import org.amelia.dsl.lib.descriptors.Host
import org.amelia.dsl.lib.util.Hosts

subsystem Common {
   val String artifact = "mcm-common"

   /*
    * The folder containing all of the Java projects (must be the same across the
        different hosts)
    */
   param String sources = "/tmp/matrices"

   /*
    * The folder containing the compiled jar files
    */
   param String assets = "assets"

   /*
    * Common dependencies
```

```
23     */
24    param List<String> classpath = #['<assets>/<artifact>.jar'];
25
26    /**
27     * All hosts, organized by host identifier
28     */
29    param Map<String, Host> hosts = Hosts.hosts("hosts.txt").toMap[h|h.identifier]
30
31    on hosts.values {
32       init:
33          cd sources
34          compile "common/src" '<assets>/<artifact>'
35    }
36 }
```

Listing B.2: *Subsystem for the Common component*

```
1  package co.edu.icesi.driso.matrices
2
3  includes Common
4
5  subsystem Strassen {
6     val String artifact = "mcm-strassen"
7     val Iterable<String> libpath = #['<assets>/<artifact>.jar'] + classpath
8
9     on hosts.get("grid0") {
10       compilation: init;
11          cd sources
12          compile "strassen/src" '<assets>/<artifact>' -classpath classpath
13
14       execution: compilation;
15          run "Strassen" -libpath libpath
16    }
17 }
```

Listing B.3: *Subsystem for the Strassen component*

B.2.2 Requirement 2 (Example)

Specify the deployment of the Strassen strategy having into account component dependencies and the computing nodes specified in the deployment diagram. The deployment must be repeated ten times sequentially and retry on failure.

In this case we can reuse the specifications from Example B.2.1 and specify only a custom deployment strategy that meets the requirement. The sequential deployments can be achieved using a `for` statement, and the retry-on-failure feature can be implemented using the `RetryableDeployment` utility class.

```
1  package co.edu.icesi.driso.matrices
2
3  import org.amelia.dsl.lib.util.RetryableDeployment
4
5  includes co.edu.icesi.driso.matrices.Strassen
6
7  deployment CustomDeployment {
8     val utility = new RetryableDeployment
9
10    for (i: 1..10) {
11       utility.deploy([
12          start(true)
13       ], 2)
14    }
15 }
```

Listing B.4: *Custom deployment strategy for the Strassen component*

B.3 Practical Exercises

Use Amelia to specify the necessary files to solve the following deployment requirements. Please measure the time you spend while solving each exercise. After finishing the exercises, you will be given a questionnaire to evaluate a set of success factors regarding Amelia.

Exercise 1: Specify the deployment specification for the hybrid-multiplication strategy having into account component dependencies and the computing nodes specified in the deployment diagram B.3.

Exercise 2: Specify the deployment for the block-reduce strategy having into account component dependencies and the computing nodes specified in the deployment diagram B.2.

Exercise 3: Specify the deployment for the N-matrices strategy having into account component dependencies and the computing nodes specified in the deployment diagram B.4. Please remember that each processor component in the N-matrices strategy requires a multiplication service; for this exercise, please reuse the specifications from Exercise 1.

Exercise 4: Define two more deployment requirements for the case of study and explain how you would solve them using Amelia.

B.4 Questionnaire for Evaluating Amelia

1. For each exercise, please indicate the time you spent (in minutes) specifying it.
   E1. _____ E2. _____ E3. _____ E4. _____
2. Please describe the difficulties you experienced developing this workshop.

   _____

   _____

   _____
3. How much time have you been working as a professional Software Engineer? _____
4. Please describe how much experience you have deploying software.

   _____

   _____

   _____
5. Please describe how much experience you have with Amelia.

   _____

   _____

   _____

Please indicate your agreement or disagreement with the following statements by selecting only one square. The left-most square indicates that you **strongly agree**, while the right-most square indicates that you **strongly disagree**.

Functional Suitability

6. All concepts and building-blocks for solving problems in the software deployment domain can be expressed in Amelia. □—□—□—□—□—□
7. Amelia is an appropriate and useful tool for deploying software. □—□—□—□—□—□

Usability

8. The language elements are understandable (*e.g.*, language elements □—□—□—□—□—□

can be understood after reading their descriptions).

9. The concepts and symbols of the language resemble the terminology of the deployment domain, are learnable and rememberable (*i.e.*, learning easiness, easiness for developing deployment specifications). □—□—□—□—□

10. AMELIA helps users achieve their tasks in acceptable development times. □—□—□—□—□

11. AMELIA is appropriate for the deployment of the type of software you work on. □—□—□—□—□

12. AMELIA has useful language elements to control the actual deployment operations (*e.g.*, language elements can be selected and put into practice easily, actions are undoable, error messages that explain recovery methods are available for controlling the deployment operations). □—□—□—□—□

13. AMELIA has a concise syntax that allows expressing deployment operations in short specification files. □—□—□—□—□

Reliability

14. AMELIA prevents making errors in deployment specifications. The language constructs helps the user to avoid mistakes. □—□—□—□—□

15. AMELIA includes the right elements and correct relationships between them (it prevents unexpected interactions between its elements). □—□—□—□—□

Maintainability

16. AMELIA is composed of discrete components such that a change to one component has minimal impact on other components. □—□—□—□—□

Productivity

17. The development time of writing software deployment specifications □—□—□—□—□

is improved.

18. AMELIA helps to improve the productivity of system deployment. □—□—□—□—□

Expressiveness

19. A deployment strategy can be mapped into a AMELIA specification easily. □—□—□—□—□

20. AMELIA provides one and only one good way to express every concept of interest. □—□—□—□—□

21. Each AMELIA construct is used to represent exactly one distinct concept in the application domain. □—□—□—□—□

22. The language constructs correspond to significant application domain concepts. AMELIA does not include domain concepts that are not important. □—□—□—□—□

23. AMELIA does not contain conflicting or ambiguous elements. □—□—□—□—□

24. AMELIA is at the right abstraction level for writing deployment specifications, such that it is not more complex or more detailed than necessary. □—□—□—□—□

Integrability

25. AMELIA can be integrated with other languages used in the software development process, such as using already developed libraries. (*e.g.*, language integrability with other languages). □—□—□—□—□

Appendix C

PASCANI Grammar Definition

PASCANI grammar definition.

```
grammar org.pascani.dsl.Pascani with org.eclipse.xtext.xbase.Xbase

import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://www.eclipse.org/xtext/xbase/Xbase"

generate pascani "http://www.pascani.org/dsl/Pascani"

Model
   : ('package' name = QualifiedName ->';'?)?
     imports = XImportSection?
     typeDeclaration = TypeDeclaration?
   ;

TypeDeclaration
   : MonitorDeclaration
   | NamespaceDeclaration
   ;

MonitorDeclaration returns Monitor
   : extensions = ExtensionSection?
     'monitor' name = ValidID
     body = MonitorBlockExpression
   ;

ExtensionSection
   : declarations += ExtensionDeclaration+
   ;

```

```
ExtensionDeclaration
    : ImportEventDeclaration
    | ImportNamespaceDeclaration
    ;

ImportEventDeclaration
    : 'from' monitor = [Monitor | QualifiedName]
      'import' events += [Event | ID] (',' events += [Event | ID])* ->';'?
    ;

ImportNamespaceDeclaration
    : 'using' namespace = [Namespace | QualifiedName] ->';'?
    ;

MonitorBlockExpression returns XBlockExpression
    : {MonitorBlockExpression} '{' (expressions += InternalMonitorDeclaration)* '}'
    ;

InternalMonitorDeclaration returns XExpression
    : VariableDeclaration ->';'?
    | ConfigBlockExpression
    | EventDeclaration
    | HandlerDeclaration
    ;

NamespaceDeclaration returns Namespace
    : 'namespace' name = ValidID body = NamespaceBlockExpression
    ;

NamespaceBlockExpression returns XBlockExpression
    : {NamespaceBlockExpression} '{' (expressions += InternalNamespaceDeclaration)* '}'
    ;

InternalNamespaceDeclaration returns XExpression
    : VariableDeclaration ->';'?
    | NamespaceDeclaration
    ;

VariableDeclaration returns XExpression
    : {VariableDeclaration}
      (writeable ?= 'var'|'val')
      (=> (type = JvmTypeReference name = ValidID) | name = ValidID) ('=' right =
          XExpression)?
```

```
71    ;
72
73  ConfigBlockExpression returns XBlockExpression
74    : {ConfigBlockExpression} 'config' '{' (expressions += XExpressionOrVarDeclaration
          ';'?)* '}'
75    ;
76
77  HandlerDeclaration returns Handler
78    : 'handler' name = ValidID
79      '(' params += FullJvmFormalParameter (',' params += FullJvmFormalParameter)* ')'
80      body = XBlockExpression
81    ;
82
83  EventDeclaration returns Event
84    : 'event' name = ValidID 'raised' (periodical ?= 'periodically')? 'on' emitter =
          EventEmitter ->';'?
85    ;
86
87  EventEmitter
88    : eventType = EventType 'of' emitter = XExpression (=> specifier =
          AndEventSpecifier)?
89    | cronExpression = XExpression
90    ;
91
92  enum EventType
93    : invoke
94    | return
95    | change
96    | exception
97    ;
98
99  AndEventSpecifier returns EventSpecifier
100   : OrEventSpecifier
101     (
102      {AndEventSpecifier.left = current}
103       operator='and' right = OrEventSpecifier
104     )*
105   ;
106
107 OrEventSpecifier returns EventSpecifier
108   : SimpleEventSpecifier
109     (
110      {OrEventSpecifier.left = current}
```

```
111       operator='or' right = SimpleEventSpecifier
112      )*
113    ;
114
115 SimpleEventSpecifier returns EventSpecifier
116    : (below ?= 'below' | above ?= 'above' | equal ?= 'equal' 'to')
117      value = XExpression (percentage ?= '%%')?
118    | '(' AndEventSpecifier ')'
119    ;
120
121 CronExpression
122    : lsymbol = ' '
123        seconds    = CronElement
124        minutes    = CronElement
125        hours      = CronElement
126        dayOfMonth    = CronElement
127        month   = CronElement
128        dayOfWeek     = CronElement
129        (year      = CronElement)?
130      rsymbol = ' '
131    ;
132
133 CronElement
134    : CronElementList | IncrementCronElement | NthCronElement
135    ;
136
137 /*
138  * Options L and W of the Quartz scheduler are only supported
139  * in cases were they are found alone (by means of rule ValidID).
140  */
141 CronElementList
142    : elements += RangeCronElement (',' elements += RangeCronElement)*
143    ;
144
145 IncrementCronElement
146    : start = TerminalCronElement ('-' end = TerminalCronElement)? '/' increment =
           TerminalCronElement
147    ;
148
149 RangeCronElement
150    : TerminalCronElement ({RangeCronElement.start = current} '-' end =
           TerminalCronElement)?
151    ;
```

```
152
153 NthCronElement
154    : element = TerminalCronElement '#' nth = TerminalCronElement
155    ;
156
157 TerminalCronElement
158    : expression = (IntLiteral | ValidID | '*' | '?')
159    ;
160
161 IntLiteral
162    : INT
163    ;
164
165 XLiteral returns XExpression
166    : XCollectionLiteral
167    | XClosure
168    | XBooleanLiteral
169    | XNumberLiteral
170    | XNullLiteral
171    | XStringLiteral
172    | XTypeLiteral
173    | CronExpression
174    ;
```

Appendix D

AMELIA Grammar Definition

AMELIA grammar definition.

```
grammar org.amelia.dsl.Amelia with org.eclipse.xtext.xbase.Xbase

import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate amelia "http://www.amelia.org/dsl/Amelia"

Model
   :  'package' name = QualifiedName ->';'?
      importSection = XImportSection?
      typeDeclaration = TypeDeclaration?
   ;

TypeDeclaration
   :  SubsystemDeclaration
   |  DeploymentDeclaration
   ;

DeploymentDeclaration
   :  extensions = ExtensionSection?
      'deployment' name = ID body = XBlockExpression
   ;

SubsystemDeclaration returns Subsystem
   :  extensions = ExtensionSection?
      'subsystem' name = ID body = SubsystemBlockExpression
   ;

```

```
ExtensionSection
    :  declarations += ExtensionDeclaration+
    ;

ExtensionDeclaration
    :  DependDeclaration
    |  IncludeDeclaration
    ;

IncludeDeclaration
    :  'includes' element = [TypeDeclaration | QualifiedName] ->';'?
    ;

DependDeclaration
    :  'depends' 'on' element = [TypeDeclaration | QualifiedName] ->';'?
    ;

SubsystemBlockExpression
    :  {SubsystemBlockExpression} '{' (expressions += InternalSubsystemDeclaration)* '}'
    ;

InternalSubsystemDeclaration returns xbase::XExpression
    :  VariableDeclaration ->';'?
    |  OnHostBlockExpression
    |  ConfigBlockExpression
    ;

VariableDeclaration
    :  {VariableDeclaration}
       (writeable?='var'|'val'|param?='param')
       (=> (type = JvmTypeReference name = ValidID) | name = ValidID) ('=' right =
           XExpression)?
    ;

ConfigBlockExpression returns xbase::XBlockExpression
    :  {ConfigBlockExpression} 'config' '{' (expressions += XExpressionOrVarDeclaration
        ';'?)* '}'
    ;

OnHostBlockExpression
    :  'on' hosts = XExpression '{' (rules += RuleDeclaration)* '}'
    ;

```

```
RuleDeclaration
   :  name = ID ':'
      (=> (dependencies += [RuleDeclaration | QualifiedName] (',' dependencies +=
          [RuleDeclaration | QualifiedName])*)? ';')?
      (commands += XExpression)*
   ;

CdCommand
   :  'cd' directory = XExpression (=> initializedLater ?= '...')?
   ;

CompileCommand
   :  'compile' source = XExpression output = XExpression
      (=> '-classpath' classpath = XExpression)?
      (=> initializedLater ?= '...')?
   ;

RunCommand
   :
      'run' (hasPort ?= '-r' port = XExpression)?
      composite = XExpression '-libpath' libpath = XExpression
      (=>
         hasService ?= ('-s' | '--service-name') service = XExpression
         hasMethod ?= ('-m' | '--method-name') method = XExpression
         (=> hasParams ?= '-p' params = XExpression)?
      )?
      (=> initializedLater ?= '...')?
   ;

TransferCommand
   :  'scp' source = XExpression 'to' destination = XExpression
   ;

EvalCommand
   :  (=> 'on' uri = XExpression)? 'eval' script = XExpression
   ;

CustomCommand
   :  'cmd' value = XExpression (=> initializedLater ?= '...')?
   ;

CommandLiteral
   :  CdCommand
```

```
112     |  CompileCommand
113     |  CustomCommand
114     |  EvalCommand
115     |  RunCommand
116     |  TransferCommand
117     ;
118
119 RichString
120     :
121        {RichString} (expressions += RichStringLiteral)
122     |  (
123           expressions += RichStringLiteralStart
124           (expressions += XExpression (expressions += RichStringLiteralMiddle
                expressions += XExpression)*)
125           expressions += RichStringLiteralEnd
126        )
127     ;
128
129 RichStringLiteral
130     :  {RichStringLiteral} value = RICH_TEXT
131     ;
132
133 RichStringLiteralStart
134     :  {RichStringLiteral} value = RICH_TEXT_START
135     ;
136
137 RichStringLiteralMiddle
138     :  {RichStringLiteral} value = RICH_TEXT_MIDDLE
139     ;
140
141 RichStringLiteralEnd
142     :  {RichStringLiteral} value = RICH_TEXT_END
143     ;
144
145 XLiteral returns xbase::XExpression
146     :  XCollectionLiteral
147     |  XClosure
148     |  XBooleanLiteral
149     |  XNumberLiteral
150     |  XNullLiteral
151     |  XTypeLiteral
152     |  XStringLiteral
153     |  CommandLiteral
```

```
154     |  RichString
155     ;
156
157 terminal RICH_TEXT
158     :  "'" ('\\' . | !('\\' | "'" | '<' | '>') )* "'"
159     ;
160
161 terminal RICH_TEXT_START
162     :  "'" ('\\' . | !('\\' | "'" | '<') )* '<'
163     ;
164
165 terminal RICH_TEXT_MIDDLE
166     :  '>' ('\\' . | !('\\' | "'" | '<') )* '<'
167     ;
168
169 terminal RICH_TEXT_END
170     :  '>' ('\\' . | !('\\' | "'" | '<') )* "'"
171     ;
172
173 terminal STRING
174     :  '"' ( '\\' . /* ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') */ | !('\\'|'"') )* '"'?
175     ;
```

Bibliography

[1] ISO/IEC 25000:2014 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. Information technology standard, ISO/IEC, 2014.

[2] Apache Foundation. Apache TuSCAny. `http://tuscany.apache.org`. Accessed: 2015-09-11.

[3] Hugo Arboleda, Andrés Paz, Miguel Jiménez, and Gabriel Tamura. A Framework for the Generation and Management of Self-Adaptive Enterprise Applications. In *Colombian Computing Congress (10CCC)*, pages 1–10. Colombian Computing Society, 2015.

[4] Hugo Arboleda, Andrés Paz, Miguel Jiménez, and Gabriel Tamura. Development and Instrumentation of a Framework for the Generation and Management of Self-Adaptive Enterprise Applications. *Ingenieria y Universidad*, 21(1), 2016.

[5] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. *Technical concepts of component-based software engineering.* Defense Technical Information Center, 2000.

[6] Mario Barbacci, Mark Klein, Thomas Longstaff, and Charles Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1995.

[7] Michael Beisiegel, Henning Blohm, Dave Booz, Jean-Jacques Dubray, Adrian Colyer Interface21, Mike Edwards, Don Ferguson, Jeff Mischkinsky, Martin Nally, and Greg Pavlik. Service component architecture. *Building systems using a Service Oriented Architecture. BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, white paper, version*, 9, 2007.

[8] Nelly Bencomo, Robert B. France, Betty H. C. Cheng, and Uwe Aßmann, editors. *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, volume 8378 of *Lecture Notes in Computer Science*. Springer, 2014.

[9] Per Nikolaj D Bukh and Raj Jain. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling, 1992.

[10] David Chappell. Introducing sca. *Avalaible at http://www. davidchappell. com/writing/Introducing_ SCA. pdf*, 2007.

[11] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.

[12] John W Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, 10(1):7–29, 2005.

[14] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *annals of telecommunications-annales des télécommunications*, 64(1-2):45–63, 2009.

[15] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Bradley Schmerl, Dennis B. Smith, João P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*, pages 1–32. Springer, 2013.

[16] Alan Dearle. Software deployment, past, present and future. In *2007 Future of Software Engineering*, pages 269–284. IEEE Computer Society, 2007.

[17] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[18] Jérémy Dubus. *Une démarche orientée modele pour le déploiement de systemes en environnements ouverts distribués*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2008.

[19] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. *SIGPLAN Not.*, 48(3):112–121, September 2012.

[20] Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, et al. Ultra-large-scale systems: The software challenge of the future. *Software Engineering Institute*, 1, 2006.

[21] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[22] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1(3.4), 1993.

[23] Oscar González. *Monitoring and Analysis of Workflow Applications: A Domain-specific Language Approach*. PhD thesis, Universidad de los Andes, 2010.

[24] Richard S Hall, Dennis Heimbigner, and Alexander L Wolf. A cooperative approach to support software deployment using the software dock. In *Proceedings of the 21st international conference on Software engineering*, pages 174–183. ACM, 1999.

[25] International Business Machines Corporation. IBM WebSphere Application Server Feature Pack for SCA. `http://www-03.ibm.com/software/products/en/sca`. Accessed: 2015-09-11.

[26] Miguel Jimenez, Angela Villota, Norha M. Villegas, Gabriel Tamura, Laurence Duchien, et al. A framework for automated and composable testing of component-based services. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the*, pages 1–10. IEEE, 2014.

[27] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4):1505–1526, 2015.

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

[29] Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.

[30] H. Lee, J. Kim, S. J. Hong, and S. Lee. Processor allocation and task scheduling of matrix chain products on parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):394–407, April 2003.

[31] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[32] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[33] Metaform Systems, Inc. Fabric3. http://www.fabric3.org. Accessed: 2015-09-14.

[34] Peter G Neumann. Principled assuredly trustworthy composable architectures. *Final report for Task*, 1, 2004.

[35] Linda Northrop, Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, et al. Ultra-large-scale systems: The software challenge of the future. Technical report, DTIC Document, 2006.

[36] Oracle Corporation. Oracle Tuxedo. http://www.oracle.com/technetwork/middleware/tuxedo. Accessed: 2015-09-11.

[37] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012.

[38] Pat Shepherd. Oracle sca - the power of the composite. *Avalaible at http://www. oracle. com/technetwork/topics/entarch/whatsnew/oracle-sca-the-power-of-the-composi-134500. pdf*, 2009.

[39] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association.

[40] Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, João P. Sousa, Basil Becker, Mauro Pezzè, Gabor Karsai, Serge Mankovskii, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*, pages 108–132. Springer, 2013.

[41] Arie van Deursen and Paul Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, 10(2):75–92, March 1998.

[42] Norha M Villegas. *Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems*. PhD thesis, University of Victoria, 2013.

[43] Norha M. Villegas, Gabriel Tamura, Hausi A. Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems 2*, volume 7475 of *LNCS*, pages 265–293. Springer, 2013.

[44] Tianyin Xu, Yang Chen, Lei Jiao, Ben Y Zhao, Pan Hui, and Xiaoming Fu. Scaling microblogging services with divergent traffic demands. In *Proceedings of the 12th International Middleware Conference*, pages 20–39. International Federation for Information Processing, 2011.

[45] Qin Zhu. *Adaptive root cause analysis and diagnosis.* PhD thesis, University of Victoria, 2010.