Tesi di Laurea Magistrale

Self-Adapting System

Relatore:
Chiar.mo Prof.
Walter Cazzola

Candidato:
Nicole Alicia Rossini
Matricola 772006

Anno Accademico 2011–2012
Acknowledgments

This work is the conclusion of two and a half very long years. In this period many things happened, some things changed and (luckily) some things are still the same. I met people, shared moments and worked on different projects, with ups and downs, with many and different people around.

My first thanks are due to my family for their support and their patience. Even when they don’t have a clue about what I’m talking about or just think that I’ve lost my mind.

I also want to thank professor Cazzola for his guidance and his faith in me and his pushing to get always better results from me. I also want to thank professor France and his students Sai and Philippa for choosing to collaborate with us: this was a great opportunity for me and a challenge too.

To my friend Eleonora, for being there at noon, afternoon, midnight and night listening to everything that comes to my mind, all the time, for when I was in full pre-exam crisis, for when I say technical stuff she isn’t due to understand, for when I’m just bored.

To my pac-man mate Christian, thank you for giving me your time and your trust and for choosing to take that project with me: I couldn’t have done it without you and not with the same great result.

To my lab mates for the last year Diego, Edoardo, Stefano, Laura, Matteo and Marius, for sharing our common destiny with laughter... but now please stop misunderstanding everything I say (they know what I mean).

Thanks also to Giancarlo for Mousiké, we had our fun after all.

Thanks to all my other mates, with whom I never had the pleasure to work with – a special mention to "the girls": Alessia, Daniela, Deborah, Chiara, Francesca and Laura.

Also thank you to those who adopted me somehow here and there along the way: Flora, Marco, Alu (all right, Gabriele), Giulia, Omar, Valentina.
Thanks to those who were with me in the previous three years and stood the test of time, as this was just the conclusion of a longer path: Ricky and Leo... I wish I could say Luca, too. I want to write his name here, because I did not forget.

To all those of you who entered my life after 17.07.11, both those who welcomed me in the group and those who believed in me making me part of all this – the J-Family is amazing. You are so many I can't name you all here, I'll mention only four groups: the "sisters" Monica M., Sara T. and Sara L.; the "live" group Lau & Andrea, Barbara & Ciro; Elisabetta and Monica R. - the greatest welcome ever; the staff members Max, Joe and Matteo.

Uh, yes, thanks to the Comelico porter Salvo, otherwise he won't let me in anymore.

Sorry if I forgot to mention somebody, it wasn't intended. Thank you all.
Contents

1 Introduction 7

2 Background 11
   2.1 Autonomic Computing and Self-* properties 11
      2.1.1 Self Adaptive Software 16
   2.2 Model-driven evolution 20
   2.3 Models@run.time 24

3 The system 27
   3.1 Architecture 27
      3.1.1 Overview of the approach 27
      3.1.2 UML diagrams generation 28
         Reverser 29
      3.1.3 Diagram modification 31
      3.1.4 Operators and mappings 32
      3.1.5 Applying the changes 33
         JavAdaptor 35
   3.2 UML diagrams used 39
   3.3 Operators 41
      3.3.1 Class diagram operators 41
      3.3.2 Sequence diagram operators 44
      3.3.3 Activity diagram operators 45
      3.3.4 Precedence among operators 47
CONTENTS

3.4 Mappings

3.4.1 Mappings for class diagram operators

3.4.2 Mappings for sequence diagram operators

3.4.3 Mappings for activity diagram operators

3.4.4 Tying together sequence diagrams and activity diagrams

4 Case study

4.1 Requirements for the Train Management System and the Railway Simulator

4.1.1 Requested evolution

4.2 The study

4.2.1 Study with class and sequence diagrams

4.2.2 Study with class and activity diagrams

5 Discussion

6 Conclusions

A JavAdaptor modifications
Chapter 1

Introduction

The Cambridge English Dictionary defines *evolution* as "a gradual process of change and development".

Software evolution *per se* describes the process of progressive change and enhancement, with new versions or releases of the software, made available to users, as appropriate [LFRK01].

All software systems need to evolve: studies [LFRK01] [KNMB02] pointed out that software maintenance and evolution can take up to 80% of a system lifetime. Furthermore, many systems under systematic evolution have proved to outlive the period for which they were originally conceived.

The need of software evolution arises from the continuously changing environment, the context, where programs are immersed and the evolving reality they rule on. Lehman wrote in 1980 [Leh80]: "Programs do not suffer from wear, tear, corrosion or pollution, but they need to change when their current behavior is found to be wrong, inappropriate or too restricted". Assuming that a certain program was correctly developed (no mistakes in requirement analysis, neither errors in coding) to fulfill its given requirements, it should be natural to think that it would not need further changes, as it should already respond to all the needs of its users. The reason why a program needs to evolve is that the programs requirements are changing over time, the context of the program itself changes and the reality it is in charge to rule is evolving.

So a program need to evolve in order to keep being useful. The first law of program evolution about *Continuing Change* [Leh80] states:

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process con-
times until it is judged more cost effective to replace the system with a recreated version.

Then software evolution is strictly connected to the evolution of the purpose of the software itself, i.e. to solve a real-life problem in a changing environment. Knowing that there will be such evolution, one technique for solving this problem is to try and anticipate in which ways the system requirements might evolve. As a consequence, appropriate mechanisms can be included in the program design and implementation to directly accommodate and support such possible changes, with the pros and cons of this approach \cite{Caz09,KF05}. However, it is logical to think that not all possible evolution, as accurate as possible the analysis might be, can be foreseen and yet including it all in the program logic would lead to code pollution \cite{CazAD}. These techniques are applied, but still it is proven that most of the expense and technical complications of software evolution comes from changes not anticipated in the original design \cite{KF05}.

The research field of Unanticipated Software Evolution (USE) studies mechanisms to adapt a software system to unanticipated needs. Recognizing the inevitability of unanticipated changes in requirements, the approach aims to develop general evolution mechanisms that do not depend on prior analysis of change in requirement and consequent introduction of custom-specific evolution hooks for each developed application.

Nowadays, software systems are constantly growing in complexity and they are more and more interconnected through the internet. Furthermore some of these services became essential in our lives and our society depends more and more on them. These kind of systems are grouped under the name of non stop systems . Non stop systems might be eternal software systems or generally on-line 24/7 high-scale environments. An eternal software system is a continuously-running software system that can never be stopped. Examples include financial transaction processors, telephone switches, air traffic control, power grids, nuclear plants, military defense installations where high-availability is not only extremely important, but even vital. A sub-class of these applications are life-critical systems, which if halted might cause serious injury to people, severe damage to equipment or environmental harm. On-line 24/7 high-scale application include instead all those other services where high availability is very desirable for the success of the application itself. We can list in this category cloud computing ecosystems, social networks, chat systems, online gaming services and other web services and applications of any kind.
In the last few years innovative systems developed as the composition of software services available over the Internet appeared. Recent surveys [MPP+12] confirmed that the new trend in system development is pointing towards what is called the Future Internet:

The Future Internet constitutes the vision of a future ubiquitous, federating and all-encompassing infrastructure that will overcome structural limitations of the current Internet in terms of scalability, mobility, flexibility, security, trustworthiness and robustness. The Future Internet will culminate in the convergence of the physical ("Internet of Things") and digital worlds into a completely new reality, where richer real-time and mobile communication modalities will arise. Services will play a key role as building blocks, providing abstraction and encapsulation, not just of data storage, processing and networks, but also of devices, connectivity, physical goods and applications, content and associated intellectual property, and even physical presence.

Web services will be more and more diffused: availability and adaptability of these systems will become an open issue. The complexity of the resulting systems will also be a problem for the evolution of such systems.

The research on self-managing systems sets in this scenario. We need system which can handle their own complexity, i.e. we need to lighten the burden on developers and administrator to let the new services escalate: "there is a compelling need for developing open systems which can be adapted to new and unforeseen usage and deployment scenarios. Hiding away complexity must be coupled with – in a systematic fashion – the dynamic adaption of these software elements" [BS09].

According to [NDG+08] modern complex software systems should not have separate cycles for development and execution, and they should be developed under the assumption that they are going to change. They should have a first-class representation of itself available to enable change. These changes could be performed by humans, other systems, but also by the system itself. This first class representation is called in [NDG+08] runtime model. Also change itself should be represented as first-class, high-level entity. This idea has been applied in different applications [WGN11] [VBG+10]. The self-representation implies the use of reflection, which is also known to be a mechanism to achieve self-awareness in a system [NDG+08].

In this thesis we present a fine-grained semi-automated system for runtime software adaptation in java systems. It was conceived to be as flexible as pos-
sible in guiding software evolution through model modification.

Java was chosen as target language because many web services, also commercial ones, run on the Java virtual machine. Its virtualization model is a winning feature for system interoperability.

In this first thesis we sought a concept-proof of correctness and a feasibility study for our FGA system. This system can extract models from a running Java program (using its source code), let the developer modify the model representation of the application and it automatically propagates these changes to the running instance without altering its state. For this first thesis about this project we assumed the agent of the change to be a human developer, but once models are correctly producing runnable bytecode a software agent could be taken into account.

This project was developed in collaboration with the Colorado State University (USA). Professor Robert France’s group cooperated with professor Cazzola and me for the development of the FGA system, especially for what models are concerned.

This thesis is organized as follows: we first give a background on the related research areas in 2 then we illustrate the general functioning of our system in 3.1 and we present the involved tools; in 3.3 and in 3.4 we show in details how we modify running applications; a case study follows in 4 conclusions and future works are presented in 5.
Chapter 2

Background

2.1 Autonomic Computing and Self-\textsuperscript{*} properties

At the beginning of the XXI century in the IT field was noticed that computer systems and networks were becoming on one hand essential in everyday life and on the other hand too complex to be manually handled. Mainsah affirmed in 2002

\[\text{Systems are becoming} \] So complex, indeed, that unless there is a paradigm shift in the way computers are designed and managed, the shortage of skilled people able to administer them will be so severe that it will make Moore’s Law\textsuperscript{1} appear quite irrelevant. \hfill\text{(Mainsah in [Mai02])}

and he compared the situation to the crisis in telephony in the 1920s: the telephone system at the time worked thanks to manual switchboards, but with the phone usage kept on growing at a rate that analysts predicted that there would not be enough trained operators to work the switchboards in a short time. A major development in switching technologies was necessary to allow further expansion of the telephone network.

In both cases it was recognized that automation was necessary to sustain the workload: human operators were not enough anymore to cope with the increasing demand.

\textsuperscript{1}Moore’s Law is the observation that over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. The period often quoted as "18 months" is due to Intel executive David House, who predicted that period for a doubling in chip performance (being a combination of the effect of more transistors and their being faster).
The first appearance of the term *autonomic computing* appeared in IBM's manifest of autonomic computing [IBM01]. The word autonomic was borrowed by the biology field: the autonomic nervous system is that part of the peripheral nervous system that acts as a control system functioning largely below the level of consciousness, and controls visceral functions. It affects heart rate, digestion, respiratory rate, salivation, perspiration, pupillary dilation and all other automatic responses of our body. While most of its actions are involuntary, some, such as breathing, can be controlled also by the conscious mind. The autonomic nervous system relieves the conscious part of our brain to deal with these low level, but vital activities, and it is also a defensive mechanism as it prevents us to accidentally stop breathing or forgetting to make our heart beat.

In IBM's manifest it was pointed out how in the evolution of humans and human society automation has always been the foundation for progress. So IBM suggested that complex computing systems should also have autonomic properties, that is, should be able to independently take care of the regular maintenance and optimization tasks, thus reducing the workload on the system administrators. In the same way that the autonomic nervous system does, autonomic computing should operate autonomously on daily operations and emerge at a conscious level, i.e. to require the system administrator attention, only on rare cases, when the system needs help or needs to perform a peculiar operation where human guidance is strictly necessary.

An autonomic system is intended by IBM [IBM01] as any type of system, from a single processor, to a single server, up to a distributed system and (mostly) to larger, higher-level systems. The key concept is that the principle of autonomic computing must govern all such systems, i.e., at some level, they must be able to manage their own processes.

These higher level systems where described as system that possessed eight key elements or characteristics:

- To be autonomic, a computing system needs to "know itself" and comprise components that also possess a system identity
- An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions
- An autonomic computing system never settles for the status quo — it always looks for ways to optimize its workings
- An autonomic computing system must perform something akin to healing — it must be able to recover from routine and extraordinary events that

\footnote{sometimes addressed as involuntary nervous system}
A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection.

An autonomic computing system knows its environment and the context surrounding, its activity, and acts accordingly.

An autonomic computing system can not exist in a hermetic environment.

Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden.

Two years later (2003), Kephart and Chess published another article, redefining the concept of autonomic computing [KGC03]:

The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7.

Starting from the idea of self-management, the authors defined the four main properties of self-managed systems:

- **Self-conﬁguration**: many systems (web services in particular) are made of different systems of different vendors. Installing and conﬁguring them to work together is a time-consuming operation and requires the administrator to be an expert of all the different systems. The goal of a self-conﬁguring system is to conﬁgure itself according to a speciﬁed policy, i.e. the user should specify what kind of behavior the system should show and not how to obtain the desired behavior.

- **Self-optimization**: complex middleware may have hundreds of tunable parameters that must be set correctly for the system to perform optimally, yet few people know how to tune them. Usually these systems coexist in larger systems and the performance of one of them affects the performance of the other ones. The goal of a self-optimizing system is to ﬁnd the optimal conﬁguration providing the best service, under the current conditions of the whole system.

- **Self-healing**: system failure happens. Identifying, tracing, and determining the root cause of failures in complex computing systems is a time-consuming and expensive task. The goal for a self-healing system is to detect, diagnose and repair localized problems.
CHAPTER 2. BACKGROUND

- **Self-protection**: we constantly need to protect systems from malicious attacks and inadvertent cascading failures. The goal of a self-protecting system is on one hand to defend itself against large scale failures caused both by attacks and major failures, on the other hand to anticipate problems based on early reports from sensors and take steps to avoid or mitigate them.

These four properties gave birth to what are now called the self- properties or self-CHOP. From these initial four properties many others derived. The following properties are considered essential in order to achieve the self-managing objectives [HSD06]:

- **Self-aware**: a system must be aware of its internal state.
- **Self-situated**: a system must be aware of current external operating conditions.
- **Self-monitoring**: a system must be able to detect changing circumstances.
- **Self-adjusting**: a system must be able to adapt accordingly.

In literature many other self- properties have been defined [BJM05 CR10], we can cite self-tuning, self-organization, self-evaluation, self-adaptation, self-evolution... to which we can add environment-awareness [ScC05], but what all these properties have in common is that they describe systems which can operate on themselves and are grouped under the name of self-managed systems [KM07].

Still referring to the autonomic computing, Kephart added in 2005 a survey on research goals as the objective was still far to be reached [Kep05], even though new project was started like the Hewlett-Packard’s Adaptive Enterprise initiative and Microsoft’s Dynamic Systems initiative on the industrial side and many other research project started in the academic field (see [ST05] for a full list of active project in 2005).

He distinguished these main research areas for the autonomic computing:

- Within the autonomic element branch of the research framework:
  - **Specific autonomic elements**: Research directed towards improving the self-managing capability of specific components such as databases, storage systems, servers, etc.
  - **Generic autonomic element technologies**: Research on technologies that are generally applicable to autonomic elements, including planning, modeling, forecasting, optimization, etc.
2.1. **AUTONOMIC COMPUTING AND SELF-PROPerties**

- Generic autonomic element architectures, tools, and prototypes. Research on the internal structure of autonomic elements, tools to help create autonomic elements, and reference implementations of autonomic elements built with these tools.

- Within the autonomic systems branch of the research framework:
  - Autonomic system technologies. Research on generic technologies that entail interactions among multiple autonomic elements to achieve system-level goals, including problem determination and remediation, automated provisioning, workload management, automated installation and configuration, integrity management, etc.
  - Autonomic system architectures and prototypes. Research on system-level architectures that effectively govern interactions among autonomic elements, and prototypes that assemble the many other pieces described in this framework to demonstrate systems with improved self-management relative to existing systems.
  - Autonomic system science. Research on fundamental science of large-scale autonomic computing systems, addressing questions of learning, stability, control and emergent behavior in multiagent systems, and also addressing questions of how to quantify the degree of self-management in systems.

- within the human interaction branch of the research framework, I distinguish two sub-branches:
  - Human studies. Research on present and future interactions between human administrators and other users and self-managing systems, to determine what interfaces and other modes of interaction are most effective.
  - Policy. Research on methods for eliciting high-level policies from people, representing and appropriately transforming those policies within autonomic systems, and managing behavior with respect to those policies.

In 2010 a review [DSNH10] of the efforts made in ten years pointed out that the self- properties are the most widely recognized feature of autonomic systems. Efforts in different directions have been made over the years and a successful field for interest was the evolutionary research, but we still need a comprehensive systems theory for adaptive systems. The author attempts to give a description of the perception of autonomic systems nowadays:
An autonomic system functions as part of a wider environment and exists to fulfill some externally defined purpose. In describing the system, we must not lose sight of this purpose, suggesting that we capture its requirements and constraints within the description. These in turn provide an envelope within which we can adapt the system. Adaptation is a dynamic process, not simply a functional response to a change in some variables. There may be many acceptable choices, from which we choose a particular one to exhibit. The different choices may emphasize one system aspect over another. The choices, and the way they vary over time, provide different dynamics through the space of possibilities, a formulation similar to those physicists use with dynamic systems.

He concludes by saying that the autonomic field might have been fallen victim of excessive or unrealistic expectations, but anyway good results have been achieved in different areas; the next step in research should focus on integrating solutions and choosing solutions from the range of possibilities explored until now, i.e. he auspices more efforts will be made in autonomic systems engineering.

### 2.1.1 Self Adaptive Software

In 1997 a definition of Self Adaptive Software (SAS) was provided in a DARPA[^3] Broad Agency Announcement on Self Adaptive Software (BAA-98-12) by Laddaga:

Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

(…) This implies that the software has multiple ways of accomplishing its purpose, and has enough knowledge of its construction to make effective changes at runtime. Such software should include functionality for evaluating its behavior and performance, as well as the ability to replan and reconfigure its operations in order to improve its operation. Self Adaptive Software should also include a set of components for each major function, along with descriptions of the components, so that components of systems can be selected and scheduled at runtime, in response to the evaluators. It also

[^3]: The U.S.A. Defense Advanced Research Projects Agency
requires the ability to impedance match input/output of sequenced components, and the ability to generate some of this code from specifications. In addition, DARPA seek this new basis of adaptation to be applied at runtime, as opposed to development/design time, or as a maintenance activity.

In the first part of this definition we see that Adaptive Software is intended here as a software that can reconfigure itself when it evaluates that its own performance is not optimal. To do so it needs to know how to test its own behavior and how to change its configuration according to the test result. Laddaga himself in [LR04] comments this definition by adding:

The key aspects of this definition are that code behavior is evaluated or tested at runtime, that a negative test result leads to a runtime change in behavior, and that the runtime code includes the following things not currently included in shipped software:

1. descriptions of software intentions (i.e. goals and designs) and of program structure;

2. a collection of alternative implementations and algorithms (“a reuse asset base”).

With the second part of the definition and the later extension of its meaning Laddaga moves the definition towards software built in components which can be chosen, substituted, scheduled according some rules defined in an evaluator. The different modules are already included in the software (see point 2) and it is the evaluator concern to activate one of the different components it has at its disposal. So the adaptation here is limited to a choice among algorithms the software already has and the problem is moved to the area of self-evaluation: what parameters must the software consider? What kind of knowledge does it need? Which are the rules for the choice?

A difficult problems for Self Adaptive Software is the problem of self-evaluation. It is often not clear that we can evaluate functionality and performance at runtime. (Laddaga and Robertson)

Laddaga and Robertson in [LR04] propose a classification of problems according to their difficulty of self-evaluation:

- constructive: a constructive application is an application whose purpose is to construct a solution to reach a goal. They present as an example a
software that must construct a plan towards a goal. The self-evaluation here is somewhat embedded in the process, as to construct a possible route the program needs also to evaluate if the route is valid and if it is better than an alternate route, so we can evaluate the plan based on its merits, in comparison to constraints and goals known when the program is written. These are easy problems for the self-evaluation criteria.

- analytic with ground truth available
- analytic without ground truth

But this definition is limited to the problem itself. These types of rules are problem-specific and must be coded and inserted in each system, even though general approaches to handle them exist. These rules handle the system response to events happening in the domain ruled by the system.

Still Laddaga and Robertson [LR04] present two general paradigms for SAS:

- The plan paradigm (plan - execute - monitor - revise): treats an application as a dynamic planning system. The application first plans its actions, then executes those actions. The plan would include both tasks that evaluate how well the plan is executing and the algorithmic tasks that are the core of the application. The plan would treat computational resources such as hardware, communication capacity, and code objects (components) as resources that the plan can schedule and configure. The plan itself is available for inspection, evaluation, and modification. Replanning can occur at runtime in response to a negative evaluation of the effectiveness of the plan, or its execution.

- The control paradigm: treats an application like a control system. Evaluation, measurement, and control systems are layered on top of the application, and manage reconfiguration of the system. So the runtime software is treated like a factory, with inputs and outputs, and a monitoring and control facility that manages the factory. This kind of software is also said to execute closed loops.

Two years after the first definition was given by DARPA, Oreizy et al. give another definition of Self Adaptive Software [OGT'99]:

Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.
2.1. AUTONOMIC COMPUTING AND SELF-* PROPERTIES

The stress in this definition is on the context where the application runs in and the software is supposed to modify its behavior according to any input and configuration it is in.

Some points are also made about what should be considered when designing a self-adaptive software:

- **events that triggers adaptation**: under which conditions a program should undergo adaptation? A software can be designed to adapt itself after subsystem failures, another one might try to improve its performances in response to bad statistics of response time, another one could be designed to incorporate new behavior during runtime. Adaptation must be targeted to the desired scope. Different kind of events can obviously integrated into the same system, but it is reasonable to think that the adaptation logic would be different according to the type of event the system is facing.

- **adaptation paradigm**: a open-adaptive system or closed-adaptive system. Differently from Laddaga et al. [LR04] the focus here is not on the architecture used, but it is on the possibility of producing or inserting new behaviors inside the system. The definition they give are
  - A system is *open-adaptive* if new application behaviors and adaptation plans can be introduced during runtime.
  - A systems is *closed-adaptive* if it is self-contained and not able to support the addition of new behaviors.

The second type of system is more limited in its choices and can only adapt in response to events that were considered during the software design phase, whether the first type is more elastic and can be reconfigured to react to unforeseen situations.

- **system autonomy**: a system can adapt with different degrees of autonomy, depending on the difficulty of the task at hand and the desired degree of control wanted, degrees can vary from fully automatic, self-contained adaptation to human-in-the-loop.

- **frequency of adaptation**: in designing the adaptation mechanism it needs to be considered how often the adaptation is expected to take place and choose an appropriate policy: opportunistic, continuous adaptation to lazy, as-needed adaptation.

- **system knowledge**: chose what kind of information the adaptation system must recieve and level of accuracy it needs.
- *cost-effectiveness*: adaptation introduces overhead costs, so it should be motivated by a gain in effectiveness and/or performance, i.e. the benefits gained from a change must outweigh the costs associated with making the change. This depends also on the frequency of adaptation.

Especially by distinguishing between closed-adaptive and open-adaptive software, Oreizy et al. show that they are moving from self-adapting systems to self-evolving systems. Adaptation here is not conceived merely as a change in a program configuration, but it is conceived as the ability of the software to learn new solutions.

### 2.2 Model-driven evolution

Model-driven evolution has its root in model-driven engineering (MDE).

Model-driven engineering is a paradigm shift in the field of software engineering\cite{B05}. Its core idea is to move from code-centric to model-based practices. In \cite{GSO04} the authors write:

> The software industry remains reliant on the craftsmanship of skilled individuals engaged in labor intensive manual tasks. However, growing pressure to reduce cost and time to market and to improve software quality may catalyze a transition to more automated methods. We look at how the software industry may be industrialized, and we describe technologies that might be used to support this vision. We suggest that the current software development paradigm, based on object orientation, may have reached the point of exhaustion, and we propose a model for its successor.

In MDE programs are viewed as a composition of models, rather than a composition of objects, where the model is the abstraction of the object. Whether the basic concept of object orientation is "Everything is an object", in MDE the basic concept becomes "Everything is a model".

Bézivin makes a distinction on the two approaches by considering their essential relations \cite{B04}. In the object oriented paradigm the two main relations are:

- *instanceOf*: all objects are instances of the same class
- *inheritsFrom*: classes can inherit their behavior from other classes

while in the MDE approach Bézivin singles two main relations out:
2.2. MODEL-DRIVEN EVOLUTION

- \textit{representedBy}: everything can be captured in a model
- \textit{conformantTo}: each model is written in the language of its metamodel

The main difference of the two approaches is summarized by these relations. In the first case, the focus is on the code organization, which is represented by its classes. But in object-oriented programs, separation of concerns is not always possible, the phenomena of crosscutting concerns is well known \cite{HL95, KR03}.

The basic use of a meta-model is that it facilitates the separation of concerns. When dealing with a given system, one may observe and work with different models of this same system, each one characterized by a given meta-model. When several models have been extracted from the same system with different meta-models, these models remain related and, to some extent, the reverse operation may apply, namely combination of concerns.

The concept of model is wide.

\textbf{Model-driven architecture (MDA)} is the most widely known MDE variant, promoted by the Object Management Group (OMG) and used in industrial context. MDA may be defined as the realization of MDE principles around a set of OMG standards like MOF, XMI, OCL, UML, CWM, SPEM, etc. There is also a highly complementary trend currently building within the industry toward the realization of these MDA standards in the Java platform (i.e., standard mappings of platform-independent models to platform-dependent models, where the platform-dependent model is the Java platform) \cite{Poo01}.

In \cite{B05} many known approaches are seen as models: traces can be seen as a model of a particular execution; platforms might be defined as models in order to have a description of the concepts of platform dependence or independence for automation purposes; model elements could be models and model transformations too; the source code itself could be seen as a model of the system.

Models depict graphically or textually a system's structure and behavior from a certain viewpoint and at a certain level of abstraction. This is desirable, because typically complex system description can be better managed through multiple models, where each captures a different aspect of the solution. Models can be used not only horizontally to describe different system aspects but also vertically, to be refined from higher to lower levels of abstraction. At the lowest level, models use implementation technology concepts. Multiple interrelated models require significant effort to ensure their overall consistency.

In addition to vertical and horizontal model synchronization, the burden of other activities, such as reverse engineering, view generation, application of patterns, or refactoring, can be reduced through automation. Many of these activi-
ties are performed as automated processes that take one or more source models as input and produce one or more target models as output, while following a set of transformation rules. We refer to this process as model transformation.

The key concept of model-driven development [GLR'02] are transformations. The idea is to develop software by transforming abstract models into more concrete models and eventually into code that typically runs on top of a software platform.

Metamodelling is a common technique for defining the abstract syntax of models and the interrelationships between model elements, which are essential for defining automated transformations. Generally we can distinguish three different architectural approaches for defining transformations:

- **Direct model manipulation**: (sometimes referred to as pull): access to an internal model representation and the ability to manipulate the representation using a set of procedural APIs
- **Intermediate representation**: exporting of the model in a standard form, typically XML, so an external tool can transform it
- **Transformation language support**: a language provides a set of constructs for explicitly expressing, composing, and applying transformations

Model-driven engineering does not only introduces new ideas for program development, but it also requires a new style of evolution [DVW07].

In [Sne07] is stated that the main problem with software evolution in the MDE perspective is how to keep the description of the system synchronized with the system itself, i.e. how to synchronize the code with the model, when the system is changing rapidly and significantly.

Two main approaches to solve this problem are individuated:

- **top-down approach**: this is the implied goal of model-driven software evolution: the model is changed and the changes are automatically propagated to the real code. This assumes some kind of automatic transformation between the higher level description of the system and the lower one. The prerequisite to really applying such an automatic transformation is that the modeling language is closely related to the programming language, i.e. the higher level description of the software is not far removed from the lower one.
- **bottom-up approach**: the changes are made to the low level description of the system, i.e. to the code itself, and are then propagated by means
of reverse engineering techniques to the upper level description. This approach ensures that the model is always a true description of the system itself. However, here too, for this to work, the modeling language must be closely related to the programming language. All of the constructs in the programming language must have some equivalent in the modeling language otherwise they will be distorted.

But in [DyWO7] many other aspects are taken into account. First of all the author starts from the firm belief that software evolution is concerned with the complete life cycle of software systems, so model-driven software evolution should be a continuous process. Then he considers the fact that platform migration is a sporadic case in traditional development processes, but he considers the fact that model-driven engineering can lead to a lock-in the abstractions and generator technology adopted when the project was started. Since an MDE platform hardwires many more architectural and design decisions than a traditional development platform, platform evolution is a requirement for MDE. For these reasons he stresses out multiple dimensions of evolution for MDE:

1. **regular evolution**: the modeling language is used to make the changes; the development platform, that is, the set of domain-specific and general-purpose languages, is fixed.

2. **meta-model evolution**: changes are required to the modeling language to improve its expressivity. Such changes may require migration of models.

3. **platform evolution**: the underlying infrastructure, such as the code generators and the application framework, is required to change, because of new requirements in the target platform. Existing models may remain unaffected by such changes, if the modeling language abstracts over the specifics of the target platform.

4. **abstraction evolution**: new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain. After introducing new languages, the old system should be migrated to make use of it.

In the survey is also stressed out that there are three problematic areas in interaction:

- **Interaction between models and code**: models need to interact with the code defining the rest of the application. In particular, when incrementally introducing models in a legacy system, models need to interact with legacy
code. Modeling languages do not necessarily cover all corner cases. It is not uncommon in such cases to customize generated code, a disaster for maintainability.

- **Interaction between models**: since separate models are used to define different parts of a system, their integration requires model interaction, possibly between different modeling languages. Models at different layers of the software stack need to interact, e.g. a user interface model refers to a data model.

- **Interaction with the development environment**: the development and build environment needs to be aware of models and modeling languages and provide the same level of support as for regular languages.

All of these aspects need to be taken into account when designing evolutionary mechanisms for MDE. Many approaches exist, but the development presents a wide gap between the industrial and the academic world. The research about a common view and general techniques is still open.

### 2.3 Models@run.time

Research on model-driven engineering has mainly focused on the use of models during software development. This work has produced relatively mature techniques and tools that are currently being used in industry and academia to manage software complexity during development. Research on models@run.time seeks to extend the applicability of models and abstractions to the runtime environment, with the goal of providing effective technologies for managing the complexity of evolving software behavior while it is executing.

France classifies models in two wide families [FR07]:

- **Development models**: these are models of software at levels of abstraction above the code level. Examples of development models are requirements, architectural, implementation and deployment models. MDE research has tended to focus on the creation and use of these models.

- **Runtime models**: these models present views of some aspect of an executing system and are thus abstractions of runtime phenomena. A growing number of MDE researchers have started to explore how models can be used to support dynamic adaptation of software-based systems.
2.3. MODELS@RUN.TIME

As is the case for software development models, a run-time model is often created to support reasoning. However, in contrast to development models, run-time models are used to reason about the operating environment and runtime behavior, and thus these models must capture abstractions of runtime phenomena. Different dimensions need to be balanced, including resource-efficiency (time, memory, energy), context-dependency (time, location, platform), as well as personalization (quality-of-service specifications, profiles). The hypothesis is that because models@run.time provide meta-information for these dimensions during execution, run-time decisions can be facilitated and better automated. Thus, researches of models@run.time anticipate this technology will play an integral role in the management of self-adaptive systems.

The problems targeted by the models@run.time community are multi-faceted and thus tackling them requires expertise from a variety of research areas. A diverse set of researchers and practitioners with a broad range of expertise, including MDE, software architectures, reflection, self-adaptive systems, validation and verification, middleware, robotics and requirements engineering, takes part of the research.

An annual conference with the same name is held regularly since 2006 on the topic.
Chapter 3

The system

3.1 Architecture

In this section I’m going to present the architecture of the system and the tools involved.

3.1.1 Overview of the approach

Imagine we have a program running on a standard JVM and we need to update it without stopping it, either because we need to or we prefer not to.

The studied approach is based on the JavaDaptor tool, which can replace a running java class with another version of it. This tool works at a low level, requiring a compiled class in input and a connection to the running JVM. So, in this case, the Java source code is the interface for updating the application.

FGA extends the JavaDaptor approach by replacing the source code by more abstract models as the interface for modifying the run-time behavior of the program.

The chosen models for code modification are UML diagrams, for this work they are restricted to

- class diagrams
- sequence diagrams
- activity diagrams

The process is semi-automated as the developer is required to modify only the diagrams and to provide only a very small set of java instructions, mostly
 library calls, assignments and choose parameters. The changes made on the
models are automatically propagated to the source code, which is compiled and
fed to the reloading mechanism.

So the process is made of six main steps which can be repeated in an infinite
loop, whenever the application needs further updates:

1. Generate UML diagrams from source code
2. Let the user modify the diagrams
3. Express changes with a set of operators
4. Map those operations to functions and apply them to the original source
code
5. Compile the modified classes
6. Load the updated classes in the JVM

### 3.1.2 UML diagrams generation

At the right abstraction level, models can be used to present aspects of a soft-
ware program that can be changed in a manner that shields a developer from
extraneous details in the source code, and thus helps developers better focus
their development effort.

In order to be able to modify the code in the process, models need to be
complete and to be as faithful as possible to the code. Moreover, to every update,
models need to be updated too. So a tool (Reversoft [CPGS07]) is used to reverse
engineer the baseline source code to produce those models. This ensures that
the model is a faithful representation of the running program. Because of this,
models are generated also when the running program is first started: rather
than use the design model produced at design time, the implemented source
code is reverse engineered. This helps to overcome the well-known gap between
design information and code whose presence would jeopardize the feasibility of
the whole approach and, moreover, it renders the approach feasible also when
the design information are not available, as for systems developed with the agile
methodology.

The diagram generation is going to take place in a separate system from
the running one, using another JVM instance and a copy of the sources; we are
using a different application instance for three main reasons:

1. we do not want to interfere with the application functionality, as we might
   need to activate branches of the program that are not currently running
3.1. ARCHITECTURE

2. to extract dynamic information ReverseR actually executes parts of the program which need to be annotated and compiled by a special java compiler (see 3.1.2 for further details)

3. we do not want to capture also extra support variables introduced by Java Adaptor (see 3.1.5)

Reverser

ReverseR is a java tool developed inside ADAPT-Lab, which bases his core functionality on top of another ADAPT-Lab technology: @java.

@Java @Java is a modified version of the java language that expands the standard java annotation mechanism. Since java 5 developer can add metadata to their sources through java annotations. Elements that can be annotated are:

- classes
- interfaces
- fields
- methods
- enumerations

so the finer granularity is the fields one, but if a developer wants to add metadata about only a subset of instructions inside a method, she must annotate the whole method, not having support at a finer granularity level.

@Java aims to extend the annotation granularity by adding two values to the java.lang.annotation.ElementType enumeration so that the developer can annotate blocks of code or expressions: ElementType.BLOCK and ElementType.EXPRESSION.

Figure 3.1.2 shows the new syntax allowed by @Java, which is different from the standard one, but, like many other java extension, produces standard bytecode; in this case @java pre-processor transforms the new annotations into standard ones and then compiles using javac. Furthermore, @java comes with an extension of the java reflective API to allow @java annotation manipulation. We are not interested here in the details (which can be found in ??), we are only mentioning the method AnnotatedBlock[] getAnnotatedBlocks() which returns the AnnotatedBlock and allows the programmer to access the java bytecode corresponding to that block. Through this feature we can extract blocks of code at will and perform bytecode manipulation on parts of method bodies.
public String operation()
{
    float a = object1.getFloatValue();
    int b = (int) Math.floor(a);

    //this is an annotated block
    @BlockAnnotation("custom annotation"){
        a = object2.doSomeComputation(b);
        b = (int) Math.ceil(b);
    }

    //this is an annotated expression
    return @ExpressionAnnotation("custom defined")(b*a);
}

Figure 3.1: Example of @java annotations: a block annotation and an expression annotation.

or we can decompile the bytecode and extract the sources for each block.

ReverseR exploits this extension of the annotation mechanism and defines its own set of @java annotation to be used for diagram generation. These annotations not only contain metadata about the diagrams, but also drive the generation process. ReverseR takes as input a set of compiled classes and a configuration file and it produces diagrams as final output, but some annotation can override the configuration settings inside their scope, so the whole generation process is highly customizable. Still, for our needs, diagrams need to be generated all at the same way, with the same configuration, and have the most complete information as possible.

ReverseR works in two phases: during the first one, the instrumentation phase, annotations are processed and substituted with ReverseR method calls through the boot library, which performs low-level bytecode injection; during the second phase the manipulated program is executed and diagrams are produced during its execution. This allows ReverseR to capture also dynamic information, which would be invisible with a static analysis of the classes.

At the moment, ReverseR is in a porting phase: the new output format will be an IBM Rational Architect XML file. The porting is dependent on another library (Ramses 2.0) currently developed at ADAPT-Lab, so its development is tightly interwoven with the library one.

It was also necessary to update the software to be better compatible with the new java versions, as it was originally developed for java 5 and never updated. In particular it was necessary to remove an external library, xerces, because the new java versions include a standard library for that calls, which created conflicts. The class search and retrieval was also improved by using another library for bytecode manipulation at a higher level: . Thanks to this library it is possible
to extract package and class information from class files without loading them in the Java virtual machine; this allowed us to be more independent from the filesystem disposition of the class files. As ReverseR needs to retrieve all the classes of the analyzed application, it based its search on search paths given in input, but it assumed them to be the root of the packages, so sometimes it built wrong class qualified names because of these assumptions and crashed. With the extraction of the qualified name from the class file this startup problem was fixed and gave more flexibility to the tool.

In the current version of ReverseR the generation algorithm is still the original one. But, to be totally suited to be integrated the FGA system, a couple of changes need to be made to the original application:

- New fields must be added to its annotations
- Labels in sequence diagrams need to be completed with the parameters values
- Values of conditions testing need to be saved

Some testing in this direction have already been made.

3.1.3 Diagram modification

Diagrams are our interface to the change. Change agents can operate on them in order to trigger the evolution process. In this case the change agent is the human developer, but it could also be a software agent working on a higher level of abstraction.

A developer using the FGA tool makes changes to the running system by performing a sequence of changes on the models. Changes are expressed by elementary operations: we could define them as units of change. These steps could be the removal of a field in a class diagram, the addition of a action node in an activity diagram, etc... Being elementary steps towards a major change, they do not guarantee that their effect is going to leave the system in a consistent state. The consistency of the system is left to the developer programming the evolution, just like is left to the developer when writing code.

As we are going to introduce in the next section (Sec.3.1.4), the operations performed on the models are expressed with a set of operators, which could look tricky and not practical to use. Because of this, we plan on having the user to interact only with a graphical representation of the models and an equivalent graphical version of the operators inside an IDE. It would be the GUI logic task to translate them in appropriate function calls.
The GUI represents then another layer of abstraction between the running system and the developer. This allows us to define more user-friendly operations. For example all the removal operations we are going to use are not cascading ones, making it repetitive and tiring for the user to delete all sub elements one by one before being able to delete the actual element she intended to. Instead, with the help of the GUI we can emulate a cascading delete operation, which will be translated with a list of single delete operations, without extending the lower level that has to interact with the code.

### 3.1.4 Operators and mappings

We are now going to formally define the change operators.

Given a java running system, we call $S_0$ its source code and $M_0$ the set of UML models representing it. We want to obtain a consistent set of models $M_1$ representing the evolved version of this system and then propagate these changes to the sources, so to produce $S_1$.

We first define $\oplus$ as the change sequencing operator.

We call $\Gamma$ the model operations to apply to $M_0$ in order to generate $M_1$ such that:

$$M_1 = M_0 \oplus \Gamma \quad (3.1)$$

and $\Gamma$ is a composition of change operations expressed with model operators $\gamma_i$, each representing an elementary unit of change:

$$\Gamma = \gamma_1 \oplus \gamma_2 \oplus \ldots \oplus \gamma_i \oplus \ldots \oplus \gamma_n \quad (3.2)$$

(for a detailed definition of $\gamma_i$ see section 3.3).

Similarly, we call $\Delta$ the changes that need to be applied to the source code to obtain the system described by $M_1$:

$$S_1 = S_0 \oplus \Delta \quad (3.3)$$

Also $\Delta$ is obtained by a composition of changes on the sources, so we can write

$$\Delta = \delta_1 \oplus \delta_2 \oplus \ldots \oplus \delta_i \oplus \ldots \oplus \delta_n \quad (3.4)$$

So $\Gamma$ and $\Delta$ are the same changes expressed on two different layers of abstraction: the first represents the changes at a model level, the latter at source
code level. The function mapping those two expressions is \( \sigma \) such that:

\[
\Delta = \sigma(\Gamma)
\]

and holds:

\[
\delta_i = \sigma(\gamma_i) \quad \forall i \in [1, n]
\]

so equation (3.3) can be written as:

\[
S_1 = S_0 \circ \sigma(\Gamma)
\]

The diagram in figure 3.2 shows the architecture here described. \( S_0 \) is the source code of the running application and \( M_1 \) is the set of models extracted with ReverseH in step 1. The first set of models is referred to also as \( M_0^0 \). In step 2 each operator \( \gamma_i \) is applied to the previous model version \( M_{i-1}^0 \) and transformed into \( M_i^0 \). We obtain a chain of intermediate models:

\[
M_i^0 = M_{i-1}^0 \circ \gamma_i
\]

and, at the same time, each change on the models must be applied to the sources too, so we obtain a corresponding chain to the previous one (3.8):

\[
S_i^0 = S_{i-1}^0 \circ \delta_i
\]

where the link between the two chains is given by \( \sigma \), see equation (3.6).

The last set of models obtained after the application of \( \gamma_n \) is the representation of a new version of our application, so we are going to call those models \( M_1 \). Likewise, after applying the last change \( \delta_n \) we obtain the new sources \( S_1 \). It is from these sources that we are going to extract the new models in the next iteration. We decided to extract the models instead of using the modified ones, to be certain to be in the same conditions at the beginning of each iteration.

In the next sections we are going to further illustrate the \( \gamma_i \) operators and the way they are mapped by \( \sigma \) on the source code.

### 3.1.5 Applying the changes

When the changes are produced by the developer inside the IDE, the mapping function is not immediately applied. The translation process is triggered once at the end of the procedure. During the modification phase the IDE records the actions applied by the programmer in terms of operators and parameters. At
the end an ordered list of operation is passed to the next phase. This avoids useless operations triggered by human errors and it does not leave the system in an inconsistent state, trusting the developer to trigger the update process only when all the diagrams have been updated and checked for consistency.

Figure 3.3 shows the workflow: once the mappings and the sources are fed as input, each change $\gamma_i$ is mapped through $\sigma$ to the corresponding code change $\delta_i$, producing new sources; this modified source code is compiled; then the modified classes are selected as input for the JavAdaptor tool, which is triggered in the last step.

When the update is completed, the new sources are once again given to ReverseR, which will produce a new set of diagrams as base for a new update cycle.
3.1. ARCHITECTURE

JavAdaptor

JavAdaptor is a tool developed at Magdeburg University. It allows class replacement and schema change in a standard Java virtual machine (JVM), where this is not possible.

In a Java system classes are loaded by the classloader subsystem. In order to avoid long program startup times, the JVM loads the required classes lazily. Moreover, if a class is already loaded, the class loading process will be aborted. This prevents us to load multiple instances of the same class, so reloading a new version of a class is not allowed.

The classloader system is made of a hierarchy of different classloaders:

- the bootstrap class loader
- the extension class loader
- the application class loader

as a further extension of the application class loader, a program can implement and use different user-defined class loaders, with custom hierarchy defined among them.

As each class is assigned to the class loader of higher level that loaded it: this creates namespaces. So multiple instances of the same class could exist in different namespaces. The only way to unload a Java class is to unload the classloader which loaded it, so loading each class with a different class loader could appear to be the solution, but this is not feasible due to the overhead cost and the fact that it breaks the uniqueness of static elements.

The solution adopted by JavAdaptor is built on top of the Java Debug Interface (JDI), so it acts like a debugger for any Java application by connecting to a running JVM at any moment.

The JDI provides explicit control over a virtual machine’s execution. The ability to suspend and resume threads, and to set breakpoints, watchpoints, … Notification of exceptions, class loading, thread creation… The ability to inspect a suspended thread’s state, local variables, stack backtrace… (Java SE Technical Documentation)

So JDI also allows a programmer to access the running state of a virtual machine. This allows JavAdaptor to access the state of each running object and save it or restore it in new instances.
CHAPTER 3. THE SYSTEM

For changing code at runtime JavAdaptor uses the HotSwap technology. Even if HotSwap is not a standard feature, nowadays it is implemented by all major certified Java virtual machines commonly used in production, including the Sun’s (now Oracle’s) official distribution, so we can call it a de facto standard and affirm that JavAdaptor can work on any java system.

HotSwap allows developers to change a method body on a program instance running in debugging mode. Exploiting this feature, JavAdaptor can manage changes in method behavior, but not yet changes in class structure.

The first problem to be solved is the one of loading a new version of the same class. The solution used is to rename the new version with a suffix that keeps track of the class version, so no class with the same name exists and they can be loaded at any time by the same classloader that loaded the original version of the class.

Having now the new version available, JavAdaptor can change all the references to the old version and make them point to the new one. This is done through HotSwap.

But the class renaming introduces a problem on the variables types: TempSensor and TempSensor_v2 are of two different types. Where the declaration is made inside a method body, the type can be changed like in fig.3.6, but this is not possible where the declaration is in another part of the class, i.e. field decla-
3.1. ARCHITECTURE

Figure 3.6: JavaAdaptor: reference updating.

To solve this, a new field of type IContainer is injected in every class on the application start-up, so there is an area of storage of a known interface available in any class.

New objects for the existing fields are stored in the new IContainer field (which can contain a list of objects) and references to fields are automatically redirected to their newer versions stored inside the container.

The last issue of this approach is solved through proxies: new classes don’t fit return type of methods. If a new version of a class must be returned than the return type is different from the one originally declared. Then, the object is wrapped in a proxy inheriting from the original version and then unwrapped.
CHAPTER 3. THE SYSTEM

Figure 3.8: JavAdaptor: proxy.

Thanks to these simple object manipulations JavAdaptor can load and replace all the local objects. Moreover, it can also create new instances for long-lived objects, copy their state in the new instances and substitute them. To do so, JavAdaptor briefly suspends the involved threads. This is a small price to pay (a matter of few seconds) to have runtime changes and it is nothing compared to a full restart. Against what is declared in [PKC’12] the newest version of the update procedure has already been made thread-safe.

As we already said JavAdaptor can connect and disconnect from any JVM in debug mode and the tool is designed to be able to reload different versions of the same class many times. The level of indirection it introduces allows multiple replacement, without adding further indirection except for the first time.

Having this tool as back-end for the update process, we can allow ourselves to be independent from when and how to execute the update: JavAdaptor will take care of these matters. Our only concern will be to feed it the right .class files.

To this purpose in mind, we had to develop a different version of this tool: in its original version it is an Eclipse plugin. Although that is very useful for application development and testing, it wasn’t suited to be included in this deployment chain, as we needed to trigger the process from an external tool and we didn’t want to be dependent on Eclipse’s project files and its file system calls.

Furthermore we saw that, even though it was designed to do so, the current implementation was not able to handle multiple projects, which inside the
3.2. **UML DIAGRAMS USED**

Eclipse IDE was not a major issue, because it could easily create and extract all the information needed every time there was a project switch, but it could be tiring to create every time using it as a library. So we introduced the notion of projects inside the tool, too.

For implementation details see A.

### 3.2 UML diagrams used

For the FGA project we focused on three different UML diagrams: class, sequence and activity diagrams. At first we focused only on class diagrams and sequence diagrams, then we expanded the work and included activity diagrams.

**Class diagrams**  Class diagrams were our first choice.

Class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. ([Peo04])

The class diagram is the most used among UML diagrams ([Peo04]) and it is essential to understand the structure of the classes that compose an application: it gives all the structural information. It is also easy to automatically produce and analyze: many tools, commercial or open source, are available. In [MRR11] is also described an algorithm to compute differences between two different versions of the same diagram (our $\delta_i$).

So class diagram was our choice for a structural diagram. To be suited to our purposes a class diagram must:

- show all classes of our target application (but we can accept them to be split into different views for usability)
- show all class members, including private ones
- show full qualified names
- show field’s type
- show method’s full signature, including parameters types and return type

As we are going to show in the mappings section 3.4, class diagrams, thanks to their static structure, are easy to map to the code they represent and changes performed to them are very easy to transfer to the sources.
**Sequence diagrams**  Class diagrams give us a full view of the class structure, but a Java program is made of class instances: objects. Each object presents different behaviors in different scenarios and a program behavior is dependent on its object’s behavior. To show and manipulate behavior, our first choice were sequence diagrams.

Sequence diagrams are listed among the interaction diagrams inside UML 2.0 specifications. They show how objects interact in single scenarios.

What was interesting about them for us is that they show method calls through the message abstraction. In a program execution control-flow statements like loops and tests activate some branches of code, while skipping others and some portions of code are also replicated. So sequence diagrams can be seen as the desired trace we want to obtain with a certain execution.

To be suited to our purposes a sequence diagram must:

- be linked to what event triggered the particular sequence
- have unique IDs for all messages
- show name and type of the object referenced in every lifeline or show a special class representation for those global classes with only static methods
- accept complex labels

In the first stage of development our sequence diagram presented UML decorations containing the source code specific to that method call. This was necessary as the mapping was not precise at line-level and required the intervention of the developer. This is no longer needed as activity diagrams were added.

**Activity diagrams** In a second stage of our work we decided to introduce activity diagrams.

Activity diagrams recall flowcharts, but they include features for parallelism and synchronization.

We decided to add them as control-flow structures could not be modeled with only the previous two diagrams. Our interest in them is focused firstly on the decision node. This allows us to model loops and tests (see sec. 3.3 and sec. 3.4 for details).

Our activity diagrams represent a single method, so we are going to have as many activity diagrams as our methods are. We need them to support:

- action nodes with name and description
3.3 OPERATORS

- unique block IDs
- stereotypes

3.3 Operators

In this section we are going to analyze in detail the operators we use to modify the diagrams. These operators embody well-defined semantics of specific forms of UML diagram changes, and thus they provide the means to map model changes to code changes. Their effect on the sources is going to be taken in analysis in section 3.4.

Even if operators work on models, our design started from code level on java systems, with mappings as final target, so operators are restricted to those UML model features which can be related to java code. Features that can not be straightly traced back to java constructions were not taken into analysis.

We recall from section 3.1.4 that, given a set of UML diagrams $M_0$ representing the running system, we apply a sequence of operations $\Gamma$ and produce a new set of UML diagrams $M_1$, i.e.,

$$\Gamma = \{\gamma_1 \circ \gamma_2 \circ \gamma_3 \circ \cdots \gamma_n\}$$ such that $M_0 \circ \Gamma = M_1$

Since models $M_0$ and $M_1$ must be consistent, $\Gamma$ must be a sequence of operations which transforms a consistent set of diagrams to another consistent set of diagrams when applied in the given order. However this condition may not hold for all the intermediate steps $M_i$ obtained by applying $\gamma_i$ on $M_{i-1}$. Each $\gamma_i$ is an elementary step that changes only one aspect of a model, so there is no guarantee that the result will be consistent at each intermediate step. Only $\Gamma$ is required to preserve the consistency of the model.

We now take a closer look at the $\gamma_i$ operators and their syntax, for every UML diagram we considered.

3.3.1 Class diagram operators

Class diagrams represent the static structure of our running application. This is the place where a developer would start from for structural changes, this is why we are going to call class diagram operators also structural operators. Our operators need to allow major and minor changes to the program structure, so they are going to work on major and minor elements inside the diagram. Operators for class diagrams work on the following class diagram elements:
CHAPTER 3. THE SYSTEM

- classes
- interfaces
- fields
- constructors
- methods

Although in UML class diagram specifications many kinds of relationships are listed, among which we can list aggregation, composition, dependency, generalization and realization, only the last two were considered meaningful to class changes.

Even if for human comprehension of the application aggregation and composition are very useful, for a developer with an update in mind they are not of primary relevancy: that kind of relationship is dependent on the class fields, so a developer would be more interested in changing the field type and having the relationship change with that update, instead of changing the association and expecting one of the fields to change type. So this kind of relationships will not be taken into account and hidden in the diagrams.

Instead, generalization and realization give information of primary importance for the classes structure. The Generalization relationship indicates that one of the two related classes is considered to be a specialized form of the other. A realization is a relationship between classes, interfaces, components, and packages that connects a client element with a supplier element. A realization relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface. So the two relationships express the concepts we need in order to include inheritance and polymorphism inside the model and we need them to be objects of the change operators to fully support the object oriented paradigm.

So the operators for class diagram will have as target these element we said up here.

The major operations a developer might want to do on a class diagram are the insertion of a new element or a deletion of an existing element. So, we have two major operators $\oplus^{CD}$ (where $CD$ stands for class diagram), which is the add operator, and $\ominus^{CD}$, which is the remove operator, designed for these two main operations.

Being the elements they operate on so different among them, these operators need to be specified according to their target element. For the add operator $\oplus^{CD}$ we then obtain:
3.3. OPERATORS

- \( \oplus_{\text{class}} \) (element_name, visibility)}
- \( \oplus_{\text{interface}} \) (element_name, visibility)
- \( \oplus_{\text{field}} \) (element_name, visibility, type, target_class)
- \( \oplus_{\text{constructor}} \) (element_name, visibility)
- \( \oplus_{\text{method}} \) (element_name, visibility, return_type, parameters, target_class)

Likewise, for its opposite operator (the remove operator) \( \ominus_{\text{CD}} \) we obtain:

- \( \ominus_{\text{class}} \) (element_unique_name)
- \( \ominus_{\text{interface}} \) (element_unique_name)
- \( \ominus_{\text{field}} \) (element_unique_name)
- \( \ominus_{\text{constructor}} \) (element_unique_name)
- \( \ominus_{\text{method}} \) (element_unique_name)

The remove operations are intended to be not cascading, i.e. the class operator can be applied only on an empty class.

With these few operators we nearly covered all the main changes a developer could want to do, except for reorganizing the relationships among interfaces and classes. As we already mentioned, we need to be able to manipulate the generalization and realization relationships. Being the two so similar, we use the same operator for both of them and we are going to distinguish which of the two is being used by looking at the type of their target elements. The operator \( \triangleright \) is the operator that adds a generalization relation between two classes. Its opposite operator is \( \triangleleft \), which removes an existing generalization relationship.

For class diagrams we initially developed a full set of operators supporting also modification and smaller operations. The set of supported operators was then shrunk to the add and remove operation to better define a base set which could be extended later. For example a modify operator was considered at the beginning, but it was cut out from the base set because it could be replaced by a remove and an add operation with content temporarily saved in memory, edited and then submitted. This does not mean that the operator in itself is useless, but it was not strictly needed for our analysis. The modify operation could be added in a second time either as a full operator or as a macro of operations at GUI level.
### Table 3.1: Structural operators in their first design: full list of operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Target</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>element</td>
<td>target</td>
<td>Package name, class name, method name, constructor name, interface name</td>
</tr>
<tr>
<td>MOVE</td>
<td>element</td>
<td>source</td>
<td>Package name, class name, method name, constructor name, interface name</td>
</tr>
<tr>
<td>DELETE</td>
<td>element</td>
<td>value</td>
<td>Name, class name, method name, constructor name, interface name</td>
</tr>
<tr>
<td>TYPE_REFINE</td>
<td>type</td>
<td>target element</td>
<td>class name, interface name</td>
</tr>
<tr>
<td>RENAME</td>
<td>target element</td>
<td>value</td>
<td>Field, method, class, interface</td>
</tr>
<tr>
<td>CH_MODIFIERS</td>
<td>target element</td>
<td>values</td>
<td>Field, constructor, method</td>
</tr>
<tr>
<td>CH_RET_TYPE</td>
<td>target element</td>
<td>value</td>
<td>Method, constructor, method</td>
</tr>
<tr>
<td>INIT</td>
<td>target element</td>
<td>value</td>
<td>Field, method</td>
</tr>
</tbody>
</table>

#### 3.3.2 Sequence diagram operators

Sequence diagrams show interactions among objects in a single use case [Fox04]. We use sequence diagrams to have information on the objects interactions. As they show objects behavior we are going to call the operators that modify sequence diagrams also **behavioral operators**.

The elements our sequence diagram operators work on are:

- lifelines
- messages

and messages can be divided into the following types:
3.3. OPERATORS

- create
- synchronous
- asynchronous

As we are talking about Java systems, all methods are expected to return, so all message calls are going to be modeled with a synchronous message. The only methods that could need an asynchronous message to be modeled are methods related to thread handling like `start()`, `wait()`, `notify()`, `notifyAll()`, `sleep()`, ...

As these methods are very few and the control flow does not get transferred, we are going to model them as synchronous methods for now. Also in the case of the method `wait()` the execution will re-start sooner or later from where it was suspended, so it does not really influence the way they are shown in the models.

So, like with class diagrams, we have an add operator $\oplus$ that is specialized for sequence diagram elements and its opposite operator $\ominus$ to remove existing entities. We then obtain:

- $\oplus_{\text{lifeline}}(\text{sequence, unique\_name, class})$
- $\oplus_{\text{message}}(\text{type, message\_id, from\_lifeline, to\_lifeline, inside\_method, label, line})$

and

- $\ominus_{\text{lifeline}}(\text{unique\_name})$
- $\ominus_{\text{message}}(\text{message\_ID})$

3.3.3 Activity diagram operators

Activity diagrams describe each method with a separate diagram. Doing this we have a clear and straight representation of each method in our application and we can reason at a very fine detail on the system behavior. Operators for activity diagram need to be as fine grained as possible to guarantee such level of control.

UML 2.0 includes many elements that can be included inside an activity diagram, but those we are going to consider are:

- action nodes
- decision nodes
• initial and final nodes

• input and output pins

What it should be noticed at once is that the transaction element does not appear in the above list of supported elements. We decided not to manipulate them directly, but to associate or incorporate them into other diagram manipulations since they are tightly related to the elements they connect and to avoid consistency checks. We could do that using the assumption that each activity diagram represent one and only one method, so certain constraint must hold:

• each diagram has a single initial node and a single final node

• there is no flow termination except for the final node

• the flow is continuous, it starts in the initial node and terminates in the final node: all forks must be joined before the diagram end

This joined to the way we designed our operators guarantee that there will be no need to operate hand-free transformations on transactions.

Operators we are going to use are very similar to class diagram operators \((3.3.1)\) and sequence diagram operators \((3.3.2)\). Like the latter they are behavioral operators, as they describe how the control flow moves from action to action.

Like for the other diagrams we designed an add operator \(\oplus^{AD}\) and a remove operator \(\ominus^{AD}\) for the different elements we are going to take into account and whose manipulation we are going to support.

Especially for the add operator we are going the GUI to give us more information than a simple distinction from action nodes and decision nodes. Especially the latter needs to be analyzed, together with outgoing transaction to identify what kind of structure it represents. With decision nodes we can model both simple tests and loops, which differ from each other because the latter has a transaction entering into itself or before itself, while the first forks the flow into different paths that all re-unite in a single flow at some point.

In its turn the flow of a loop can encounter: first the decision node and then an activity node before re-entering into the decision one; first an activity node and then a test and have the back leading transaction entering the main flow before the activity node.

Also tests can be modeled differently. The main difference is on the type of test they perform: do they evaluate a single boolean condition or do they activate an outgoing transaction according to the value assumed by a variable?
3.3. OPERATORS

If they test a boolean condition, they might have one or two branches before re-joining in the main flow, because they one might want to add behavior only when the condition is true, only when it is false, or add different behaviors in both cases. Instead, if they test the value of a variable they can start as many branches as the possible values plus a default one which jumps special behaviors and re-joins the main flow.

Any other form except these are do not present structures that can be related to code structures: they are unfeasible. Because of this, we prefer not to give too much freedom with transactions to the final user, because we want to obtain a diagram that can represent a method body, so we limit her choices by giving through the GUI pre-built blocks of decision nodes and transactions.

The add operator specializes in:

- \( \oplus \) action(method, flow, after)
- \( \oplus \) test(method, flow, after, type, branches_no, condition)
- \( \oplus \) loop(method, flow, after, type, condition)
- \( \oplus \) description(block ID, text)
- \( \oplus \) instruction(block ID, line_no, text)

Likewise the remove operator specializes in:

- \( \ominus \) action(block ID)
- \( \ominus \) test(block ID)
- \( \ominus \) loop(block ID)
- \( \ominus \) description(block ID)
- \( \ominus \) instruction(block ID, line_no)

3.3.4 Precedence among operators

Those operators we presented till here are elementary operators which perform minimal units of change on diagrams. Having split the change in such small operations we do not guarantee that each of the \( \gamma_i \) operations produce a consistent model, as one semantic change could be made of different \( \gamma_i \) operations. But at the same time we do not leave the change totally unsupervised. In order to guarantee correctness of the mappings, we need to impose a partial ordering on the operator application.
CHAPTER 3. THE SYSTEM

Ordering for class diagram operators and sequence diagram operators

On our first version of this project only sequence diagrams were considered to provide behavioral information. This is the first ordering we produced to rule the mappings with only class diagrams and sequence diagrams operators:

1. ⊕interface and ⊕class: interfaces and classes need to be added first, as they are empty containers for the other elements in class diagrams and sequence diagrams that must refer to them.

2. ⊕field, ⊕method and ⊕constructor: the static part of methods and constructor, as long as with fields, must be added before adding behavior.

3. ⊖: if a generalization relationship needs to be changed, we have to remove the old generalization before adding the new one, as we are working on Java models. Java classes can extend a single superclass, so we need to remove the existing one before adding a new one. Abstracting from Java, this behavior does not introduce errors, so we can extend it for models in any language.

4. ⊖: we can now build new hierarchies.

5. ⊕lifeline: when class diagrams are complete, we can start adding elements into sequence diagrams.

6. ⊖message: where the create messages have a higher precedence on the synchronous messages; note that messages have a progressive numeration, unique for each sequence diagram, to help the addition of new messages.

7. ⊖message: similarly to ⊖message but synchronous messages are deleted before the creational ones.

8. ⊖lifeline: removes unused lifelines.

9. ⊖field, ⊖method and ⊖constructor: removes unused elements inside classes.

10. ⊖interface and ⊖class: removes unused classes and interfaces.

Note that removal operations are all made at the end of the process and that operators for sequence diagrams are in a central position; by doing this, we guarantee that methods exist before messages are added and that messages are removed before removing methods, that is, all ⊕CD operations are the first to be executed and ⊖CD operations are the last to be executed.
3.3. OPERATORS

Ordering for activity diagram operators

When we extended our work to activity diagrams some of the mappings had to change in order to integrate two different visions of the same aspects of the code. So also the ordering was reconsidered and extended.

Some of the previous arguments still hold, because these operators need to work on a defined structure, so points from 1 to 4 are still the first operations to be performed (additions on class diagrams) and points 9 and 10 are still the last operations to be performed (removals on class diagrams).

For interaction between sequence diagrams and activity diagrams we refer to section 3.4.4, but we can say here that none of the sequence diagram mappings perform a direct code modification.

So activity diagram operators take the place of sequence diagram ones and they need to be applied between class diagram additions and class diagram removals.

Among activity diagrams operators we have strict precedence rules:

1. ⊕test and ⊕loops: as they build the control flow structure, they need to be applied first, following their ID value, which is incremental. By doing so, when nested structures are added we guarantee to add these structures from the most external to the most internal one.

2. ⊕action: we need to add action nodes inside the control flow, so to know where to add the real code.

3. ⊕description: we can add action node description when action nodes are already in place.

4. ⊕instruction: with action nodes in place we can now add lines of code inside them.

5. ⊖instruction: instruction are the first elements to be deleted, so to leave its action node empty if it needs to be deleted.

6. ⊖description: the same said for ⊖instruction holds for descriptions.

7. ⊖action: we can now delete empty action nodes, before removing control flow modifiers.

8. ⊕test and ⊕loops: control flows are the last to be deleted, with opposite order for their addition: from the most internal to the most external one.
CHAPTER 3. THE SYSTEM

3.4 Mappings

Now that we presented the \( \gamma \) operators, we are showing how they are translated by the \( \sigma \) mapping and the effect that \( \sigma(\gamma) \) has on the source code.

3.4.1 Mappings for class diagram operators

Class diagrams represent the structure of classes, hence \( \sigma \) provides a one to one mapping between the changes in the class diagram done via the defined operators and the application code structures.

The \( \oplus_{CD} \) adds its specific element into the classes. In figure 3.9 it is shown modifications applied to a very simple system with this operator and in 3.10.

\begin{itemize}
    
    \item (a) Original system
    \item (b) Effect of operator \( \oplus_{\text{class}} \)
    \item (c) Effect of operator \( \oplus_{\text{field}} \)
    \item (d) Effect of operator \( \oplus_{\text{method}} \)
    \item (e) Effect of operator \( \triangleright \)

\end{itemize}

Figure 3.9: Class diagram example: evolution of the models.

In 3.9(b) the \( \oplus_{\text{class}} \) is used to add a `NewClassB` to the system. The operation is \( \oplus_{\text{class}}(\text{NewClassB, public}) \) is mapped by \( \sigma \) into source code and its effect is shown in 3.10(a). The \( \oplus_{\text{interface}} \) has a very similar mapping: instead of the \texttt{class} keyword, \( \sigma \) uses the \texttt{interface} keyword.

The \( \sigma(\oplus_{\text{field}}) \) mapping is shown in 3.10(b). In 3.9(c) a field is added and the corresponding code is added on the first line after its class declaration. This choice was made to maintain the source code readable by humans, since in the
3.4. MAPPINGS

Figure 3.10: Class Diagram example: changes impact on sources.

Java language fields declaration can be made at any line of code, but usually fields declaration are made at class top.

The $\sigma(\textit{method})$ mapping effect is 3.10(c) for the method addition made is 3.9(d). Methods, as well as fields, can be added at any line, still, for the same choice of preserving sources readability, we decided to add methods at the class bottom. The same operator can be used to add methods into interfaces. Its mapping differs in this case as it does not write curly braces for the method body and adds instead a semicolon after the method declaration.

The last mapping is the one for the $\triangleright$ operator. It adds generalization to classes, as shown in 3.9(e). It has two possible mappings: either it uses the \texttt{extends} keyword or the \texttt{implements} keyword. The choice can be automatically computed by checking the father’s and child’s types:

<table>
<thead>
<tr>
<th>Father’s type</th>
<th>Child’s type</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>class</td>
<td>\texttt{extends}</td>
</tr>
<tr>
<td>interface</td>
<td>interface</td>
<td>\texttt{extends}</td>
</tr>
<tr>
<td>interface</td>
<td>class</td>
<td>\texttt{implements}</td>
</tr>
<tr>
<td>class</td>
<td>interface</td>
<td>_</td>
</tr>
</tbody>
</table>

Table 3.2: Rules for $\sigma(\triangleright)$ mappings

The $\ominus^{CD}$ operators mappings are the $\oplus^{CD}$ operators mappings reversed. All these operators do not perform a cascading removal of elements: the removed element must be already empty to be removable. This give us control over the deletion process. This solution however is not handy for the developer who might want to remove elements fast. The solution relies in the GUI: the interface will provide the user more sophisticated tools for the removal of composed elements, but it will transform the composed operation into sets of elementary operations.
For example to remove a whole interface the system will write a remove $\Theta_{\text{method}}$ operation for each method declaration in the interface and a $\Theta_{\text{interface}}$ operation at the end.

Mappings of these operator are still one-to-one mappings.

$\Theta_{\text{class}}$ and $\Theta_{\text{interface}}$ remove the same portion of code added by $\Theta_{\text{class}}$ in 3.10(a) which is the class or interface declaration.

$\Theta_{\text{field}}$ removes the field declaration. As no two fields with the same name can exist inside a class, the removal is straightforward. The same remark holds for method declarations and the $\sigma(\Theta_{\text{method}})$ mapping.

$\sigma(\phi)$ removes the extends or the implements asserts, following the same rules of table 3.2.

### 3.4.2 Mappings for sequence diagram operators

In a first version of this work we focused our attention only on class diagrams for structural changes and sequence diagrams for behavioral changes. The first mappings we developed do not take into account activity diagrams and information they bring to the general view. We will show here our analysis on correspondence between sequence diagram elements and code structures, then we will show how this is brought together with activity diagrams.

Sequence diagram operators are more complex than class diagrams operators. The first difference necessary to stress out is that sequence diagrams represent a dynamic interaction among objects, while class diagrams represent the static structure of its classes. Moreover, sequence diagrams represent a behavioral view on a single execution, so they do not capture a complete view of the whole system at once.

For these reasons, we do not find here one-to-one mappings, but we need to use more rules to map some elements to their respective code structures, while others do not present a direct mapping to any structure.

The $\oplus_{\text{lifeline}}$ operator which adds a lifeline into a sequence diagram. Even if in a sequence diagram a lifeline is a definite element, there is no direct correspondence in the code. Figure 3.11 illustrates the effect of an application of the $\oplus_{\text{lifeline}}$ operator on the model.

Adding a lifeline means that another object, which is not necessarily a newly created object, is part of the message exchange sequence. We could affirm that the correspondent structure to a lifeline is an object reference pointing to the newly created object. But this object does not interact with all other objects in the diagram, but only with those it receives a message from or sends a message.
3.4. MAPPINGS

(a) Original system

(b) System with a new lifeline

Figure 3.11: Sequence diagram example: addition of a lifeline into the base system.

Object it sends a message to do not need a reference to it. Only objects it receives messages from need to have a reference to the object itself, but we can not make assumptions on how these objects acquire this reference. The reference could be either a local variable inside a method or it could be stored in a field. Moreover the object could be created locally, passed as a method or constructor argument, set by a setter method, retrieved through a getter method or it could be already present in the caller object, but it hadn’t been used in the shown use case until now.

Therefore we can not fix a mapping for the $\oplus$ operator. In many of the previous cases the reference setting will appear later in the diagram, before messages are sent to the object, otherwise if it does not appear and messages are sent to the object, we trust the developer to know the reference was set somewhere before.

Figure 3.12: Sequence diagram example: addition of a creation message.

The other element it can be added to a sequence diagram is a message. Generally speaking a message is a method call that needs to be added inside
the method body. Its mapping can vary according to the type of message that is added.

@message has two different mappings depending on the kind of the added message: create or synchronous.

- @message(create, ...) adds a creation message. A creation message means a new lifeline (see fig. 3.12) is added and, as a lifeline represents an object, its corresponding mapping is a constructor call. The class of the created class is extracted and its constructor is retrieved. If the message presents a label with a syntax different from the base constructor, the appropriate constructor is chosen. The constructor call now must be inserted in the object represented by the sender lifeline, inside the method from which the message originated. In fig. 3.12 the call new ClassB() must be inserted in method m0() belonging to class ClassA. There are no further information we can extract here from this diagram alone.

Furthermore, the newly created object should be saved either in a local variable or in a field: figure 3.13 shows the two possible mappings.

```java
public class ClassA {
    private ClassB objectB;
    public void m0() {
        objectB = new ClassB();
        stmt1;
        stmt2;
        ...
        stmtn;
    }
}
```

(a) initializing a field.

```java
public class ClassA {
    public void m0() {
        ClassB objectB = new ClassB();
        stmt1;
        stmt2;
        ...
        stmtn;
    }
}
```

(b) initializing a local variable.

Figure 3.13: Example of code affection ambiguity due to a new creation message.

As a rule we could use the following: if a field of a compatible type and with the same name of the created object was added in the class diagram, we can suppose this it the same object and so we would store the new object inside that field; otherwise we create a local variable with the lifeline name and assign the object into this variable.

Note that, in this case, a call to the constructor could be added everywhere inside the code of m0() except within loops and conditionals, because we have no further information about its location inside m0(), so a good insertion point would be as first instruction.

For disambiguation of these two cases, we accept a label containing a variable declaration like we are going to do for synchronous messages.

- @message(synch, ...) adds a synchronous message, which corresponds to a
method call. We could have also asynchronous messages in java, but they
are rare and their mapping is not different from the one for synchronous
messages, so we are not going to analyze them explicitly. We are going to
use a synchronous message also for asynchronous messages meaning that
after making the call the control flows comes back to the caller method.

Let us consider the situation described in Fig. 3.14 where a new method
call to the method m1() is added to the sequence diagram in Fig. 3.12.
The method m1() must exist in class ClassB and ClassA will use objectB
and it is up to the user to guarantee its existence. In this example, the
code to add is objectB.m1() and from the sequence diagram we can evince
it must be added in the body of m0() in any position, after new ClassB().
In case of static method the rule to generate the line of code to be added
becomes in ClassName.methodName().

In a more general case, the new message is going to be added between to
messages, so the portion of code where to insert the call is more restricted.
It is necessary to track back which method calls originated those messages
and then isolate the lines of code among the two calls. For a finer level of
insertion we will move to activity diagram mappings in 3.4.3.

The last issue to deal with is parameters. Adding a message with parameters
requires to specify which parameters must be passed. A list for each parameter
made of fields and local variables (compatible with the method signature) is
presented to the user. The user can decide whether to use one of the presented
fields or add a constant value. In order to save the returned value we accept
message labels with the assignment specified, like: Type varName = method(..).
A particular case is when objects are stored in the variables representing the passed parameters: if we had a method `public void method(String parameter1){..}` the user could decide to reuse the variable `parameter1` to store a new object. We are going to accept a message label containing parameters name (\(\text{parameter1} = \text{new String()}\) or \(\text{parameter1} = \text{object1.getObject2()}\)) and treat them like any other already declared local variable, trusting the developer for that to be a legal action.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation message</td>
<td>if there is a field with the same name of the new object and it is of a compatible type then instantiate that field, otherwise create local variable.</td>
</tr>
<tr>
<td>Synchronous message</td>
<td>if the method is static then add <code>Class.method()</code>, otherwise use the lifeline name as variable name <code>objectname.method()</code></td>
</tr>
<tr>
<td>Return value</td>
<td>if the message label contains a variable assignment <code>Type varName = method(...)</code>, insert the assignment too</td>
</tr>
<tr>
<td>Parameters</td>
<td>if the method has parameters build a list for each parameter containing all compatible variables or fields; ask the user to choose one or insert a constant value.</td>
</tr>
<tr>
<td>Conditional statement</td>
<td>If a message is added it is supposed to be valid for each method call. When we will deal with activity diagrams too, we will handle also conditional statements and the message will be added only for the selected cases.</td>
</tr>
</tbody>
</table>

Table 3.3: \(\sigma\) rules for \(\oplus\) message.

Similar considerations can be done for the \(\ominus\) operator.

\(\ominus\)lifeline does not affect the code.

The removal of a lifeline means that the object represented by the lifeline ceases of being involved in the interaction described by such a sequence diagram. In general this does not mean that the object is removed from the system so any action that implies its removal is not feasible and it is not performed. Therefore, the object removal depends on its removal from the class diagram if it is a field or more in general on the removal of its creation message.

Instead the \(\ominus\)message causes the deletion of a single line of code: the one which contains the method or constructor call described by the exchanged message from the method body referred from the source object lifeline. Note that the call is univocally identified from its position inside the sequence diagram. A exceptional case is represented by multiple messages referring to the same method call, e.g. a method call inside a loop will be triggered many times and it will show in the sequence diagram as multiple messages, one for each iteration over the loop. In this case, the removal will affect all those messages at once. The developer will confirm the deletion by pointing out the line of code.
to remove, so she will be aware of the consequences of her action.

3.4.3 Mappings for activity diagram operators

Activity diagrams in our conception are a detailed representation of each method. We support modifications of activity diagrams in order to support changes in behavior.

For each method in our system we already have a generated activity diagram. For each new method inserted by the $\oplus$ method diagram instead, we have a new activity diagram made of an initial and a final node, connected by a transaction (see fig. 3.15). This is a collateral addition performed by the $\sigma(\oplus)$ mapping.

![Transaction 1](image1)

Figure 3.15: Activity diagram of an empty method

![Activity diagram with a new action node](image2)

Figure 3.16: Activity diagram with a new action node

Transactions must be addressable and it this necessary that they are linked to code. We are going to hook transactions with a @java annotation \(3.1.2\): @transaction(int id). We need to be able to address each transaction in order to add nodes in specific branches. They are different from ReverseRe annotations for transactions, which are @ControlFlow and @SimpleControlFlow, because their design does not match our needs. ReverseRe design is more general and does not make our same assumptions, because its purpose is more general then ours: we restrict an activity diagram to the space of a single method. As a result ReverseRe annotations present a more general behavior and a complicated usage, so we decided to add a more simple annotation for our hooks. Being inside a method, its body represents the flow going from the method activation to the return point. When there are no conditional statements or loops, its activity diagram is represented by an ordered sequence of action nodes. In figures 3.15 and 3.16 the addition of an action node does not alter the flow. To ReverseRe in 3.16 there are two different ControlFlows: the first going from the start node to the action node, the second going from the action node to the end node. We do not make this distinction and we count as different transactions only the branches of the main flow, which can be found inside the code as those portions
of codes inside curly brackets of control flow structures.

Addition of a new action node is performed by the \texttt{@textaction} operator. Action nodes do not have any direct representation in Java, as they are blocks of instruction performing a certain action. Therefore we need to add that information inside the code. ReverseR (3.1.2) handles them by introducing a \texttt{@textjava} block annotation. \texttt{σ(@textaction)} adds the ReverseR annotation in the specified transaction line, after the specified node (see 3.4.3). If a transaction is still empty, the node is added as first block, while the value of the after parameter is 0. All ReverseR annotation for activity diagrams have an ID attribute, so we are going to exploit that ID for our purposes.

![Diagram](image)

(a) Base diagram. (b) Base code.

| public void moveToParentFolder(String file) {
| @Transaction(id=1) {
| \texttt{@CallAction(id=1, name="Get parent’s absolute path")} {
| String path = file.getParent();
| int i = path.lastIndexOf(File.separator);
| path = path.substring(0, i);
| }
| @CallAction(id=2, name="Build new abs path") {
| String filepath = path + File.separator;
| filepath += filename + " . " + ext;
| File newFile = new File(filepath);
| }
| @CallAction(id=3, name="Move file") {
| file.renameTo(newFile);
| }
| }

Figure 3.17: Base method

In actions nodes we are going to show the source code details, but descriptions (names) can be given to nodes too to preserve the immediately of diagrams readability. The \texttt{@textdescription} operator does that. Its mapping is simple: it sets the given value to the name parameter of the \texttt{@textCallAction} annotation corresponding to the node described, an example is shown in 3.19(b) where the operator is applied to the method shown in 3.4.3.

\texttt{σ(@textinstruction)} adds a line of code inside the diagram, so \texttt{σ(@textinstruction)} introduces that line of code inside the corresponding block of code (see 3.20(b)). Lines are counted locally and their offset refers to the first line of the block they are inserted into. This is going to help us handling movements of code blocks, when
public void moveToParentFolder(File file) {
    @Transaction(id=1) {
        @CallAction(id = 1, name = "Get parent's absolute path") {
            String path = file.getParent();
            int i = path.lastIndexOf(File.separator);
            path = path.substring(0, i);
        }
        @CallAction(id=2, name="Build new abs path") {
            String filepath = path + File.separator;
            filepath += filename + "." + ext;
            File newFile = new File(filepath);
        }
    }
    @CallAction(id=4) {
    }
    @CallAction(id=3, name = "Move file") {
        file.renameTo(newFile);
    }
}

Figure 3.18: Effect of the @action operator

@description(4, "Make backup copy")

@test adds a decision node and a number of branches according to the type of decision is added. This operator can be mapped onto two types of conditional structures: if - else construct or switch construct. The first corresponds to a boolean test, the second, instead, has a variable which is being tested for some values.

In the first case (a single boolean test) \(\sigma(\@test)\) is going to add a whole statement like in \(3.4.3\) and insert the hooks for the transactions this block has created. The mapping always inserts a full if structure with both the then block and the else block, also when only one of the two blocks is used, in order to let the user add
blocks in both the branches. The operator creates only the structure, it does not insert also the code to be executed inside the if. This is because code is part of an action node, so after inserting the structure the user will specify the actions to be added inside these branches using the new transactions as anchor points for the new nodes. The [True] and [False] labels here are just for the diagram readability.

![Diagram operation](image1.png)

In the second case instead, the mapping would be the one shown in 3.4.3. The testing is carried out on the variable specified inside the decision node and labels on transaction express the value this variable must assume for its transaction to be chosen. In this case the mapping is going to be a switch statement, where a default case is always present, while other cases are added as many as the outgoing transactions are and their values are extracted from these transaction labels. This block corresponds only to a switch statement,
where all cases ends with the **break** statement. If the same behavior is desired for multiple cases a copy of the same activity node must be added to all the desired transaction.

In both cases, the condition is supplied with the operator: in the first case it will be a string representing a boolean test, in the latter it will be a string representing a variable or a method call returning a primitive type.

![Diagram](image-url)

**Figure 3.22: Effect of \( @\text{test}(\text{values}) \)**

The \( \sigma(\@\text{loop}) \) mapping can present two forms as well.

In Java there exists two types of loops: the **while** construct and the **do - while**, plus the **for** construct that we are going to consider as a special syntax of the first one. The mapping is going to insert the code blocks for the two cases as shown in **3.23**. In **3.23(c)** the structure is the one of a **while** construct: the test on the boolean condition supplied is taken before executing its content. In **3.23(d)** instead, there is the transaction 2 that is encountered before the decision node. The transaction marked as **[true]** is a dead transaction, as it is not possible to map to Java code a block with action nodes in two two different places of the same loop, so no transaction ID is associated with it.

The **for** construct is going to be shown with an UML stereotype and modeled as a **while** loop. For those constructs generated via ReverseNI we are then going to see in the stereotype the variable declaration which we are looping on and its increment. We need three different stereotypes:

- **<for:decl>** for an action node: this will represent the variable declaration during loops (usually \( \text{int } i = 0 \))
- **<for:stmt>** for a decision node: this will represent the actual loop
- **<for:incr>** for an action node: this will specify the kind of increment we want to operate onto the variable (usually \( i++ \))
3.4.4 Tying together sequence diagrams and activity diagrams

In section 3.4.2 we presented mappings for sequence diagram operators developed without information about activity diagrams. Sequence diagrams, although essential for understanding dynamic behavior, lack of information to automatize the mapping process. In their early version sequence diagrams mappings depended on UML decorations showing those portions of code where a method call could be inserted. Furthermore they did not take into account what would happen if messages were added inside loops or conditional statements and they did not allow the insertion of new structures of that kind.

With activity diagrams in the picture, we can now add, remove and therefore modify method bodies with precision.
Sequence diagrams represent the system behavior during a single execution and show which transitions are activated inside activity diagrams and help us link them together.

In figure 3.24, a simple execution is shown through a sequence diagram and two activity diagrams which represent the two methods we are going to focus on. In this sequence diagram, we added the ID of the action node messages belong to after the # symbol.

The main method executes two very simple calls on object `a` of class `ClassA`: first it calls method `a1()` then method `a2()`. Its activity diagram is the first one. Looking only at this activity diagram, we can not understand what is going to happen when `a1()` is called. We see what happens in the sequence diagram. So we can say that message `a1() #3` ties together action node #3 with the activity diagram representing method `a1()` of class `ClassA`.

Even if in this simple example the binding seems trivial, sequence diagrams help us linking the right activity diagram when the called method is a polymorphic one. Imagine that instead of method `a1()` we had method `poly(Square)`, `poly(Rectangle)`, `poly(Triangle)` in that class and we had a generic method call `poly(Figure)` inside action node #3. In this case, we would not know which of the three methods would be actually called. We could choose the right activity diagram only extracting the information from the sequence diagram of the use case we are analyzing. This could also help us building a bigger activity dia-
gram representing a whole module logic as a composition of activity diagrams of single methods.

The other thing to be noticed in figure 3.24 is that now we can easily map at which point of the method body to add a new message, having now the notion of action node mapped to a code block. Method b1() corresponds to the action node #5, while the three messages b2() corresponds to three different activations of node #6 which is placed inside a for loop.

But now the ⊕ message and ⊕ instruction (together with the ⊕ action one) mappings overlap. They both add a message call, but the first does not specify in which action node to place the call, unless it is placed inside two calls belonging to the same block, neither it can add a new action node. Furthermore sequence diagrams should not allow the developer to add only single message inside of a loop, because it would be replicated for each time the loop is repeated: either she adds it in every iteration or the operation is illegal. The same problems are shown by the ⊕ message and ⊕ instruction (together with ⊕ action).

Because all of these reasons we decided to change the mapping for the ⊕ message operator and the ⊕ message operator. They will perform the same action on sequence diagrams, but when the point of insertion is calculated the corresponding activity diagram will be presented to the developer to explicit which kind of insertion to perform on the latter. Note that when polymorphic methods are in the picture the choice of the correct activity diagram is not that trivial and we will use information extracted from the sequence diagram to perform the choice. Having the developer expressed the insertion also on one activity diagram the ⊕ message operator will be partially processed and changed to one or more ⊕ AD operators: first the method call to be inserted will be calculated, then one of the following will be used:

- ⊕ instruction: the method call is inserted in an existent action node (the simpler case)

- ⊕ action node ⊕ ⊕ instruction ⊕ ⊕ description: the method call is inserted in a new action node, which might have a description associated to

The modification of the sequence and the activity diagram must be handled by the GUI as an atomic modification, i.e. it should trigger the mapping of ⊕ message to obtain the line of source code that should be added, find the right activity diagram and let the developer finish the mapping. The GUI will then insert the process only the ⊕ AD operators in the list.

An analogous mechanism will be used for the ⊕ message operator. When a
message is deleted, the corresponding line of code will be found in the action node it belongs to and the user will be asked for a confirmation. If the action node has no further line it will be deleted with the $\Theta_{\text{action}}$ ($\Theta_{\text{description}}$ will be used too if needed), otherwise the only operator used will be the $\Theta_{\text{instruction}}$. 
Chapter 4

Case study

Our case study comes from the information systems domain and considers a train management system (TMS).

We implemented a train management system as an example of non-stop system which could benefit from our work. This type of system can not be usually stopped at will, because they need to be constantly available: they control and rule human activities which do not stop when the system is not available.

This case study a simple implementation of a more complex software, as it does not fulfill all the requirements of a system of this kind (e.g. security, reliability, availability), but it helped us understand what kind of changes might be necessary and how to model them.

The TMS of our case study is responsible for tracking trains and for controlling the traffic signals that determine whether a train should stop or proceed: it handles the policy of every traffic lights of the railway system.

We implemented also a railway simulator (RS) for the train management system, in order to simulate all the traffic signals in a realistic way.

4.1 Requirements for the Train Management System and the Railway Simulator

Trying to simulate a real-life scenario, our TMS was modeled as a server receiving network inputs by sensors scattered along the railway system and reacting to them by sending commands to its traffic lights. In order to give regular inputs to the system we implemented also a simulator for the railway system that
moved trains at a regular pace on the network, following the given rules.

Both the system have their own representation of the railway: the TMS needs it for decision making, the RS uses it to move trains correctly.

The railway is made of:

- Stations: each station has a platform for each route that flows into it or leaves from it (so to always have a free platform for an incoming train). All platforms are all connected by railroad switches, so a train can enter a station on any platform and leave on any route.

- Platforms: a platform is treated as a kind of segment as it will have a light and a sensor.

- Routes: a path between two stations only. Each route has a direction, e.g. route From station A to station B, is different from route from station B to station A.

- Route segments: Each route has a number of segments, a segment is not shared among routes. Each segment is uniquely identified.

- Journeys: a set of connected routes. Each train declares to the TMS which journey it will follow.

- Train System (Train Map): is an ensemble of stations connected by routes.

For a simple implementation we used the following rules:

- Trains start from a station belonging to the railway. Each train declares a journey and waits for the permission from the TMS before leaving its first station. When it reaches the last station of its journey it computes the opposite journey of its current one and travels back to its first station.

- A train must travel the sequence of segments on a route, i.e. no jumps.

- Only one train can be in a segment at any time. This is achieved using a sensor and traffic light: each segment has one sensor and one traffic light at the end of a segment. The sensor at the end of a segment detects trains as they leave the segment. Each segment also has a traffic light at the outgoing end. This light determines whether a train can enter the next segment or not. If the light is red then the train must stop. If the light is green then the train can proceed to the next segment. The light is green only when there is no train in the next segment and there is no train that has just left the previous segment. If there is a train in the next segment then the light in the previous segment must be red.
When a sensor detects a train leaving a segment this information is transmitted to the TMS and used to update the position of the train, turn the light red in the segment and turn the light green in the previous segment.

A train that does not obey a red light must be marked "Runaway Train". A train stops being a Runaway Train when it obeys a red light.

Processing of simulation data: Each line shows what happens in a time unit. We'll take the view that the actions taken are atomic. A train takes at least one time unit to complete travel on a segment.

So the Railway Simulator is implemented to initialize and move the trains. For each train that enters the system, it must declare its journey and have it validated. This validation is accomplished through communication of the Railway Simulator with the TMS, in this case the communication happens between each train and the TMS. For each train with a valid journey, the RS moves them on the next segment of their own journey. When sensors sense that a train has left the segment they are linked to, they signal the TMS that a train has just left their segment. In response the TMS decides which traffic lights need to switch their color and sends a message to those traffic lights.

For the TMS to be able to correctly detect when the rules governing the movement of trains have not been followed, the Railway Simulator is implemented to disobey the rules at random intervals. The initial implementation of the TMS and Railway Simulator is for passenger trains. This represents the first planned implementation for the TMS and the Railway Simulator.

4.1.1 Requested evolution

Imagine now that the owner company decides to expand its business transporting new kind of goods, which require special treatment like, for example, hazardous materials (in the next a train transporting hazardous materials is called hazardous train). When transporting hazardous materials special security rules must be attended and the TMS must be updated in order to insert these new rules to enforce the railway security. These rules change the traffic light policy when hazardous trains are travelling on a route and, as a consequence, they also introduce the concept of hazardous train into the TMS.

Rules for Hazardous Trains are:

- No other train is allowed on a route if a hazardous train is travelling on that route. If a passenger train is on a route that a hazardous route
wants to travel, the hazardous train is not allowed on the route until the passenger train has left that route.

- At most 1 hazardous train is allowed per each route
- All lights for all segments behind a train on a route must be red while the hazardous train is travelling that route.

4.2 The study

We started from the implementation of a base system (TMS1), following the requirements presented in 4.1.1. We did not implement the base version with possible updates in mind, so to make it tailored only to its initial requirements. We deliberately did not choose to use many design patterns, because we did not want a system already predisposed to easy adaptation.

When we had this tool in our hands, we started thinking how this application might evolve given a new requirement. The new requirement was to include goods trains in the railway system, originally designed only for passengers train needs, with a peculiar case of trains transporting hazardous materials. Hazardous trains require extra security rules, so the TMS must change its behavior and its policy for traffic lights settings if a train transports hazardous materials.

Given the first design we placed ourselves in the TMS developer’s shoes and imagined what changes we would want to apply to the system in order to insert these new rules.

Our first consideration was that the strategy pattern \cite{GHJV94} would be appropriate to handle different policies for the same operation (switching the traffic lights color), but it was a costly operation, as it involved heavily the program architecture. We decided to insert it anyway on the assumption that the developer would prefer to predispose the TMS also for future changes, not only for the one at hand. So our update was divided into two different updates.

1. The first one would change the program architecture and have the strategy pattern (as long with a factory) inserted in a system which externally would present the same behavior of its first version.

2. The second one would be the actual insertion of the rules for handling the hazardous train, that would exploit the newly inserted architecture.

To fully understand what was required inside the code to change we implemented both the evolution steps in TMS2(version with the strategy pattern) and TMS3(version with the hazardous train).
4.2. THE STUDY

From both the coding experience and the UML diagrams we planned our first version of our operators from which we elaborated them up to the version presented in 3.3.

While we designed the operators we tested them back on diagrams to see if the result of their application matched the planned version or if it produced an analogous or very similar result.

4.2.1 Study with class and sequence diagrams

In an intermediate phase of our work we limited our study to class and sequence diagrams. Activity diagrams were later included to solve some problems addressed by the sole manipulation of sequence diagrams.

We requested the developer to interact partially with the source code, as there were issues inserting a method call in a method body with precision without the exact knowledge of the method structure.

So, through UML decoration and the help of the GUI, when the developer inserted a new message in a sequence diagram we showed her a pop-up decoration with the lines of source code belonging to the portion of code where we calculated the insertion would be feasible and she could specify the exact line of insertion. For loops and conditional statements there were no structures that allowed their insertion, nor it was possible to insert messages inside existent conditional statements except by manually specifying it inside a decoration.

Except for this we could manage major changes also with these open issues. We are going to show here a sequence of operations for evolving TMS1 into TMS2. We remember that the change applied here is the application of the strategy pattern.

![Figure 4.1: TMS class diagram, before/after the strategy pattern reorganization.](image)

Figure 4.1 shows a portion of the class diagram before and after the change is applied. The only class of the original version affected by the change is the
CHAPTER 4. CASE STUDY

Message. The pattern impacts only on the Message class, which represents the
evaluator for the messages the TMS receives from the RS; Message implements
the Runnable interface; when a message is received the computation is threaded
to evaluate it. In the first version only two kind of messages can be received: the
journey authorization request and the notification of the train position from the
segments sensors. Message has methods to deal with these messages and chooses
(through an if statement) which method to execute on creation. In the new
version instead, Message is an abstract class and it is subclassed as many times
as the supported kind of messages; each subclass implements how to evaluate
the corresponding message and the selection is demanded to the polymorphism
mechanism. Message also acts as a factory, choosing which subclass better rep-
resents the received message.

The main changes between the two class diagrams are: new classes extend
Message, methods belonging to Message are moved in the new subclasses, con-
structors are now private and new Message object can be created only through
a static method in Message.

The following is an excerpt of the operations that have to be applied to go
from the class diagram in Fig. 4.1(a) to the one in Fig. 4.1(b).

- @class(EnteringStationMessage, public)
- @class(SensorMessage, public)
- @class(JourneyMessage, public)
- @class(EnteringStationMessage, Message)
- @class(SensorMessage, Message)
- @class(JourneyMessage, Message)
- @method(public, static, Message, {String}, Message)
- @constructor(protected, (String[]), EnteringStationMessage)
- @constructor(protected, (String[]), SensorMessage)
- @constructor(protected, (String[]), JourneyMessage)
- @method(run, public, void, {}, EnteringStationMessage)
- @method(correctPlatform, public, String, (String,String), EnteringStationMessage)
- @method(ButTrainLeaveStation, public, String, (String), EnteringStationMessage)
- @method(run, public, void, {}, SensorMessage)
- @method(checkIfRunaway, public, boolean, (String, String), SensorMessage)
- @method(updateSystemTime, public, void, (int), SensorMessage)
- @method(updateTrainPosition, public, String[], (String,String,int), SensorMessage)
- @method(run, public, void, {}, JourneyMessage)
4.2. THE STUDY

- \( \Theta \text{method}(\text{authorizeJourney}, \text{Message}) \)
- \( \Theta \text{method}(\text{changeSegmentState}, \text{Message}) \)
- \( \Theta \text{method}(\text{checkIfRunaway}, \text{Message}) \)
- \( \Theta \text{method}(\text{correctPlatform}, \text{Message}) \)
- \( \Theta \text{method}(\text{letTrainLeaveStation}, \text{Message}) \)
- \( \Theta \text{method}(\text{updateTrainPosition}, \text{Message}) \)
- \( \Theta \text{method}(\text{interpretMessage}, \text{Message}) \)
- \( \Theta \text{method}(\text{updateSystemTime}, \text{Message}) \)

These operations change the structure of the involved classes. In the first half of the list, new elements are added, while in the second half, useless structures are deleted. Between the two parts, operations for behavior modification take place. We achieve to change the program behavior through the sequence diagram operators.

We are not going to show here the details for all these changes, because the list would be too long and would not be meaningful without a full knowledge of the source code of both the TMS versions.

We are presenting here a subset of the full list of those operations focusing on a single change localized in a restricted portion of the TMS application. We show here the adaptation of the most affected part on the behavioral side, i.e., the part dealing with the message identification and evaluation (Fig. 4.2), while in the other portions of code we have old methods moved into new classes, which require many operations of insertion and deletion, but basically copy the old code in a new place.

In the original sequence diagram the RailwayIn class, which is the class that receives the messages, directly creates a Message object. The planned change foresees to use a static method (\text{getMessage(String)}) instead. This method is going to call the correct constructor to deal with the subclasses. Therefore the object that we are now passing to Thread is not a Message object anymore, but an instance of one of its children classes (see Fig. 4.2(b)).

The whole sequence diagram must change: methods that were in the Message class now are in its child JourneyMessage class, validateJourney(), addTrain(t), sendMessage(output) must be called from the journeyMessage:JourneyMessage lifeline and their old invocation must be deleted.

\(^{1}\text{Following the precedence rules given in Sect. 3.3 the operations on sequence diagrams are interleaved with those on class diagrams. For sake of comprehension we will present the operations on sequence diagrams later.} \)
Figure 4.2: The sequence diagram describing the run method of RailwayIn.

The following is an excerpt of the operations that have to be applied to go from the sequence diagram in Fig. 4.2(a) to the one in Fig. 4.2(b).

1. \( \oplus \text{lifeline} \) (journeyMessage: JourneyMessage)
2. \( \oplus \text{message} \) (synchronous, 5, RailwayIn, Message, Message m2 = getMessage(m), 33)
3. \( \oplus \text{message} \) (create, 5.1, Message, JourneyMessage, inside(5), new JourneyMessage(m), 208)
4. \( \oplus \text{message} \) (create, 6, RailwayIn, Thread, after(5), new Thread(m2), 34)
5. \( \oplus \text{message} \) (synchronous, 7, RailwayIn, Thread, after(6), start(), 35)
6. \( \oplus \text{message} \) (synchronous, 8, Thread, JourneyMessage, after(7), run(), 23)
7. \( \oplus \text{message} \) (synchronous, 8.1, JourneyMessage, SystemState, inside(8),
   boolean b = validateJourney(), 27)
8. \( \oplus \text{message} \) (synchronous, 8.2, JourneyMessage, SystemState, after(8.1), addTrain(t), 33)
9. \( \oplus \text{message} \) (synchronous, 8.3, JourneyMessage, RailwayOut, after(8.2),
   sendMessage(output), 49)
4.2. THE STUDY

4.2.2 Study with class and activity diagrams

In the second phase of our work we added activity diagram operators and mappings for sequence diagrams were modified. For the design of these new operators we started directly from the operators theoretical design, having already a
clear view on the subject matter.

Then, with the design in hand, we tested our operators on the TMS example to see if they allowed at least the same changes the sequence diagram did and what improvement (if any) they brought to the system.

We are going to show here the adaptation of the core of the change that occurs from TMS1 and TMS2 by modifying activity diagrams. The operation is the same presented in 4.2.1.

The class diagram operators are obviously the same of section 4.2.1 as the change is the same but obtained through different diagram manipulations.

In Message v1, which extends the Runnable interface, we have all the methods for handling every message supported by the systems, directed by the run method which contains the basic instructions, while the real functionality is split in private methods that acts like sub-routines; the run method calls the interpretation of the received message, then through an if statement activates one of the two methods that handle the specific message received. Its activity diagram and the one for the interpretMessage() method are shown in fig. 4.4(a).

When the message received is a journey authorization request, the run() method delegates its handling to the authorizeJourney() method, which is rep-
4.2. THE STUDY

The document contains code excerpts related to message interpretation and authorization in a system. The code includes method calls and variable assignments. It also references activity diagrams illustrating the流程 of methods run() and authorizeJourney().

Figure 4.4: Activity diagrams for TMS1 and TMS2
In the new version we extend the Message class so to have a subclass for each type of message received, so each new subclass will take one message handler method from the original class. Having only one method, it is no longer necessary to keep it a separate method from the run() message it inherits.

We start here to work on activity diagrams having already applied modifications to the class diagrams. First of all we have two new methods to be filled out: getMessage() in the Message class and the run() method in JourneyMessage class. We show here the operations to create the structure of the getMessage() method, which was kept with a nested if to give some complexity to the mappings, while in a more serious implementation the choice of the class would be made through reflective mechanisms:

- action(Message.getMessage(), 1, 0)
- description(70, "prepare local variable")
- action(Message.getMessage(), 1, 70)
- description(71, "make sure it’s uppercase")
- action(Message.getMessage(), 1, 71)
- description(72, "extract type of action required")
- test(Message.getMessage(), 1, 72, bool, 2, action.equals("SENSOR"))
- test(Message.getMessage(), 2, 0bool, 2, args[3].equals("ENTERINGSTATION"))
- action(Message.getMessage(), 4, 0)
- description(73, "update train position and traffic lights")
- action(Message.getMessage(), 5, 0)
- description(74, "update train platform and traffic lights")
- action(Message.getMessage(), 3, 0)
- description(75, "test if journey is valid")
- action(Message.getMessage(), 1, decision1)
- description(76, "return")

This list of operation will map to an empty method like shown in fig. 4.5(a).

We obtain the code shown in fig. 4.5(b) after applying insertion of lines of code:

- instruction(70, 1, "Message message = null;"
- instruction(71, 1, "msg = msg.toUpperCase();"
- instruction(72, 1, "String[] args = msg.split("\:");"
- instruction(72, 2, "String action = args[0];"
- instruction(73, 1, "message = new EnteringStationMessage(args);")
public static Message getMessage(String msg) {
    @Transaction(id=1){}
    @CallAction(id=70, name="prepare local variables"){}
    @CallAction(id=71, name="it’s uppercase"){}
    @CallAction(id=72, name="extract type of action required"){}

    if (action.equals("SENSOR")) {
        @Transaction(id=2){}
        if (args[3].equals("ENTERINGSTATION")){
            @Transaction(id=4){}
            @CallAction(id=73, name="update train position and traffic lights"){message = new EnteringStationMessage(args);}
        } else{
            @Transaction(id=5){}
            @CallAction(id=74, name="update train platform and traffic lights"){message = new SensorMessage(args);}
        }
    } else{
        @Transaction(id=3){}
        @CallAction(id=75, name="test if journey is valid"){message = new JourneyMessage(args);}
    }
    @CallAction(id=76, name="return"){
        return message;
    }
}

Figure 4.5: Mappings for method getMessage()

- @Instruction(74, 1, "message = new SensorMessage(args);")
- @Instruction(75, 1, "message = new JourneyMessage(args);")
- @Instruction(76, 1, "return message;;")

With these last operations we created the new method. Analogously we create the run() method of class JourneyMessage. Even though this might look more complex than writing actual code, these methods are built of the same action nodes of the old methods authorizeJourney() and interpretMessage(). The following matches can be found in diagrams of fig. 4.4

- block #55 = block #71
- block #56 = block #72
Having the support of the GUI, all these operations could be masked by a simple
copy&paste tool, which would insert new blocks in those two new methods
copying their structure from the existing one. With a simple mechanism as the
copy&paste we can obtain the ability of moving blocks of code wherever we
want in our diagram, provided variable names match.

With the 2 new methods written we need to modify the old run() method
in Message, because now we will execute only one of the message children. We
do not want to remove that method, because the child classes of Message need
to implement it. The solution adopted here is to remove the old content and
then add an RuntimeException that will rise in case of execution.

With this list of operations we add new content on the head of the diagram
(but we could have added it anywhere on transaction 1):

- \( \oplus \text{action}(\text{Message.run()}, 1, 0) \)
- \( \oplus \text{description}(61, \text{"raise error if used"}) \)
- \( \oplus \text{instruction}(61, 1, \text{"System.err.println(\"Empty method body.\")"}) \)
- \( \oplus \text{instruction}(61, 2, \text{"System.err.println(\"throw new RuntimeException (\"Run one of the subclasses\")\")"}) \)

With this list of operations we delete the old content:

- \( \ominus \text{instruction}(22, 1) \)
- \( \ominus \text{instruction}(21, 1) \)
- \( \ominus \text{instruction}(20, 2) \)
- \( \ominus \text{instruction}(20, 1) \)
- \( \ominus \text{description}(22) \)
- \( \ominus \text{description}(21) \)
- \( \ominus \text{description}(20) \)
- \( \ominus \text{action}(22) \)
- \( \ominus \text{action}(20) \)
- \( \ominus \text{action}(21) \)
- \( \ominus \text{loop}(\text{decision1}) \)

The final result is shown in fig. 4.7.

The other methods in the Message class can now be deleted. We do not
list here all the removal operations, because it would not add anything new:
4.2. THE STUDY

Figure 4.6: Mappings for method run() of class Message

the removal operation would resemble the removal operations we just per-
formed here on the run() method. Just like these operations for the creation of JourneyMessage, analogous operations are made for the creation of the other two child classes SensorMessage and EnteringStationMessage.

At the end of all the operations on the method content, the class diagram op-
eration will perform the final removal of methods that are now useless as we just moved the functionality in other classes.
Figure 4.7: TMS2: Method run() of class Message
Chapter 5

Discussion

About UML usage  In the proposed approach the evolution starts from the models, thus stimulating the user to plan the evolution on diagrams, which leads to a cleaner and more rational architecture. The decision on basing our work on UML models was based on the large diffusion it has both on the industrial side and the academic side. It is the de-facto standard for modeling object oriented applications.

Empirical experiments [DAB08] have proved that using UML during software maintenance and evolution increase the functional correctness of changes, solution quality and code quality when developers are not familiar with the system, in spite of a not significant increase in time spent on the evolutionary process. We further reduce such time by automating the propagation of the changes to the code. In other surveys it emerged that almost anybody works with UML diagrams, even though not everybody uses it with the same skill and their knowledge varies on different levels, most developers know well only a couple of diagrams and do not use all its features.

UML is not enough to describe any kind of application, sometimes specific models must be developed for special cases, but it is always a good starting point. In our case, we applied UML to a very fine grained level of detail and we used it to represent code structures. This lower-level use of UML allows us to represent any application, as these models are closely related to the source code.

It might be argued that having diagrams as detailed as ours are we might loose focus on the abstraction power of our diagrams. Usually diagrams are used to abstract from code for having a better understanding at first glance, while our conception of them might look to be pointing in the opposite direction. But
our diagrams can be seen as an intermediate step towards higher levels of abstraction. These diagrams are a representation of our source code and we could say that they are abstractions with no loss of information. Starting from them more abstract diagrams could be generated by hiding (or losing) details. For example, in our activity diagrams code details in action nodes could be hidden, obtaining only descriptive tasks; then, thanks to the information stored in sequence diagrams, we could compose our single-method activity diagrams with diagrams representing other methods, obtaining a new diagram with a wider scope. New views could be built starting from this low level UML diagrams.

**About operators** The defined operators allow modification of any part of the source code through diagram manipulation. Only a small set of structures require manual code insertion in activity diagrams. In any case, the interface presented to the developer (the set of generated diagrams) is complete, meaning that the developer can apply any modification she might desire without directly editing the source code.

Those part requiring manual insertion or modification of java code inside action nodes are local variables, **try-catch** blocks and synchronize blocks. We expect to find a diagram representation for these structures in a future expansion of the FGA system by introducing new UML diagrams and by introducing new types of operators on the already included ones.

The limited set of operation we showed in this thesis is very promising to be the minimal set of operators needed for achieving full flexibility. We hope a formal proof of this set being the actual minimal set can be found or that a refutation can lead to the definition of the minimal set.

**About automatic propagation** The automatic propagation is another strong point of our architecture: in a context where an application needs to be updated as fast as possible, it allows the user to focus on the planning with the awareness that the changes on the sources will be automatically performed and it ensures the code to be as close as possible to its model representation.

To support the automatically propagation of the changes from the model to the code we have to fill the abstraction gap between code and models [CPGS07, UAN09]. This gap is a constant problem when developing with models: an initial representation of the application to be developed is constructed on its models, but when the coding phase starts and new requirements, better solutions or error correction are applied the models become fast outdated.

In our case such gap would make the automatic generation of the code im-
possible: missing details or different versions between the code representation and the model representation of the system would cause the code to be injected in wrong parts of the program or it would make impossible to find the injection point. We face such an issue by generating UML diagrams from specific metadata (Java annotations) put in the code when developed and maintained by the system during the evolution. Given this tight link between the models and the code we can talk of co-evolution.

About diagram generation The automatic generation of UML diagrams helps us filling the gap between the code and the models and grants us to be working with the closest possible representation of the system.

In order to perform our code propagation we need very detailed diagrams for any part and any use case of the system. This task is clear to be prohibitive to be carried out without a degree of automation on the process; hand-drawn diagrams would not cover all the systems aspects and even a small mistake could lead to huge mapping errors.

The automatic generation through ReverseR solves this issue, but opens a new one. ReverseR bases its generation process on @java annotations (metadata), which are not a standard java feature and must introduced by hand by the programmer. This task might just as time-consuming as the previous one and have the disadvantage to lead us towards non-standard java derivations.

The good news about @java annotations is they get translated into standard java annotation by the @java preprocessor and then compiled with the standard java compiler javac. Recently this project has seen new life and it is moving towards a new version.

The problem of manually annotating the code instead partially remains: new code added through the activity diagram gets automatically annotated, so if the project is developed from scratch with FGA only a minimum amount of manual annotation is necessary. On the other hand, old project should be heavily annotated before being adaptable through FGA. This situation could be solved with the development of algorithms for automatic code annotation. The harder task would be to identify action nodes with a definite semantic meaning, but coarse grained action nodes could be easily identified inside source code and the developer could split them into smaller nodes when actually needed.

About the IDE The usability of the approach is based on the interaction with an IDE, which presents the developer the set of diagrams and all the operations he can perform on the models (see Sect. 3.3). Being forced to use our IDE it is not such a heavy constriction for the final user, as she would be
using a graphical IDE to draw diagrams in any case, supposing modeling was part of the deployment chain.

The IDE also takes care to express the changes in the form of function calls, which mirror the operator definitions. The operators we defined need to be expressed in a form more suitable for an object oriented program, so, supposing to transform this operators in methods of proper objects does not change anything to their theoretical definition as long as their behavior is the same showed in 3.4.

We can affirm that as our IDE is conceived it is an intermediate layer between the developer and the evolution system, providing a more sophisticated interaction and reducing the arduousness of the approach. Having this intermediate layer means we can avoid to complicate our set of operators and their corresponding mappings, and express complex operations with a simple set of operations, thus having a better control and understanding of the process. We do not exclude anyway to develop a change operator in order to support changes to an existing element, without adding a modified copy of it and then delete its older version. This operator might be very useful for example to change visibility keywords on methods and fields, change parameters types or to rename elements, simplifying the handling of multiple copies of the same object.

About JavAdaptor  Another factor which reduces the complexity of the update is the adoption of JavAdaptor \cite{PKC12} as reloading mechanism during the system execution. Many works in literature as \cite{ZCO06} focus on the definition of states where the application can safely migrate from its original form to its evolved one. This is not our concern because we rely on JavAdaptor to replace each class keeping its state intact: no data are lost and each object in his new version immediately starts running with its old state. JavAdaptor also handles when to freeze the class for the reloading operation. The new version of JavAdaptor we possess (more advanced than the one described in \cite{PKC12}) is also able to safely handle the reloading operation in a multi-threaded environment. So the constraint about when to update the application is loosened.

Furthermore JavAdaptor gives the system subject to evolution the property of openness (see \ref{sec:openness}). It carries out the lower level work of adaptation. Our system is built on top of it as a way to guide the evolutionary process.

About granularity  The constraint about what to replace is loosened, as well. Most of times software needs to evolve because the context where it operates changes, thus leading to different requirements for the application. Software evolution is as predictable as the context where it runs, so we can state that
there will always be some unforeseen evolution. From the programmer's point of view this means that any part of the software might need to be changed. We support this type of unforeseen evolution through the generation of detailed diagrams for any part of the software.

Furthermore some changes might affect only a small portion of the application, which can be restricted to only one class at times or just few lines of code. In [CLN+09] a case study is presented, where the requested change consist in adding a time constraint for an online order. While it presents a medium complexity on the diagrams, looking at the code level it consists in adding an `if` statements in many points and providing an `orderExpired` mechanism, which could be implemented by a Java exception. Significant changes in behavior can be minimal changes in the code. Our fine granularity allows us to apply fine-grained changes (at line level) to the code and replace in the running application only the affected classes, instead of replacing a whole module or component (like in [OMT98, SM04]), making the update process faster and safest as far as the program state is concerned.

About our code-centric view  FGA has a very code-centric view. As the evolution process needs to end with the production of running code, our efforts tend to mitigate the need of fast deployment and the desirable attention to a careful planning by automating the code production and make it as close as possible to the developer's wish.

As the chosen target language was java we focused on java control structures, tailoring the mappings and the operators design on this language. But the same approach could be tailored to C++ structures, designing a new version of this project. But for C++ application a part of the cycle would be missing: the reload mechanism, i.e. the mirror tool for JavAdaptor. An accurate analysis of dynamic system update tools for C++ would be necessary before moving forward in that direction.

We focused our work on how to obtain an updated version of runnable byte-code for evolving applications. We assumed for now to be dependent on a developer. Nevertheless the developer could be seen as any type of agent with the ability to manipulate diagrams, which could also be a software agent.

Anyway this limitation is not invalidating our proof-of-concepts work and the allowed fine-grained changes possible with the supports of only these two types of diagrams makes us optimist about the future developments.
Chapter 6

Conclusions

In this thesis, we presented a model based approach to software evolution able to plan fine-grained (at statement level) adaptation on the application model and effect them on the running application without stopping it.

UML diagrams of the running application are used as its model@run.time. The system manager will plan the application adaptation through them. The adaptation is split in elementary and predefined steps (described as operators) that the framework can easily map on fixed type of changes to the code. In this the reason why it is possible to co-evolve the model and the code: different changes can be expressed as a composition of basic steps whose mapping is known. Finally the planned evolution is effectively applied to the running application through the JavaAdaptor framework [PKC++12] without stopping it.

The evolutionary mechanism is kept separate from the running application that can keep on providing its service undisturbed during the planning phase, which can require some time. Furthermore the evolutionary mechanism is generic both in its approach and its interaction with the application. The approach does not make any assumption about the changes that might occur and about which part of the program they affect and it interacts with the JVM hosting the application through a standard API. This makes the system suitable to handle unforeseen changes in any standard java application.

The only restriction on the approach is on the kind of models currently supported and therefore on the kind of changes that can be described. The first version suffered from this aspect more than the second one (the one including activity diagrams), but we believe that some improvement can be obtained by including even more UML diagrams, both for types of changes supported and for different types of abstraction provided to the developer.
With this thesis we have achieved a proof-of-concepts and a feasibility study for the FGA system. Parts of the final tool have already been implemented, while some conceptual work on possible extensions needs to be done before moving to the final implementation phase. The collaboration with Colorado State University, especially with professor Robert France, is still active and the project is alive and ready to be further expanded, as we believe in its potential.
Appendix A

JavAdaptor modifications

As we already mentioned in 3.1.5 the JavAdaptor tool was modified. From an original Eclipse plugin, we transformed it in a standalone application.

We add this Appendix A as documentation about the new version.

**Eclipse projects** The original version of JavAdaptor is deeply dependent from the Eclipse project structure and its library for handling it.

*In primis*, it expects each program to be an Eclipse project, i.e. it expects to find in the Eclipse workspace directory a project directory named after the project name and inside it:

- a src directory containing all the sources files
- a bin directory containing all the compiled classes
- a .classpath file containing the classpath settings for the project launch organized in an XML format
- a .project file containing the project settings for the build of the projects itself.

This means it also expects to have an Eclipse workspace directory.

The JavAdaptor plugin it also expects to have its own storage area inside the Eclipse workspace where to save and retrieve its own data. The path expected is « Workspace location »/.metadata/.plugins/de.ovgu.javadaptor/. In this directory we can find:

bin Log.All.txt Log.Error.txt log.properties metadata plugin.properties projectStructure.xml
Inside the bin directory there are the compiled versions of some classes (like the IContainer interface) that JavAdaptor is going to insert in the target application during the launch phase and use when a reload is needed.

Inside the metadata directory we can find the ant build.xml file used for building and launching the target application and copies of the libraries, which do not appear to be used.

The most important file is the projectStructure.xml file. It contains the full serialization of the application to update. The serialization is used for keeping a copy of the status of the application during the update and it is also used when multiple updates are performed for keeping track of classes version. As the name of the file is fixed (it is always named projectStructure.xml), if we use the same installation of JavAdaptor for updating first an application, that we are going to call App1, and then a second application (App2), when we want to update again App1 we will have lost all serialization data about App1, because data about App2 would have overwritten the previous ones. Being a tool still in a development phase, we decided to contribute by adding a project notion inside the tool too.

The other high dependency from Eclipse was the use of the Eclipse class Platform. The Platform class is the abstraction of the Eclipse platform itself and offers methods to interact with it and its file system organization. The plugin code was highly bound to this class because calls like Platform.getLocation() were scattered around the whole application code.

The last dependency was the use of the Eclipse GUI functions to display few alert boxes. Alert boxes are displayed by the JavAdaptor plugin to notify the user when the connection, reload and disconnection operations triggered by the user are finished or when an error arises.

Configuration file   Being independent from Eclipse means that no assumption about where the JavAdaptors file will be. So, first of all we need a JAVADAPTOR_HOME environment variable to tell us where it has been installed.

At this location we can find a configuration file named conf.init where all the other settings can be stored. In figA is shown the testing configuration, pointing at the JavAdaptor_indip project. The project was developed inside the Eclipse IDE only as personal choice of editor, it could have been done in any other IDE of choice, being Eclipse independent. The configuration file could be enriched at will, if other fixed settings will be necessary in the future.

Inside the configuration file A after the first debug parameter JVD_HOME that prints the value used as home, we can see two variables JVD_RESOURCES_DIR and
Any update to JVD_HOME won't affect the system, update the JAVADPRODUCT_HOME env variable instead.

Tue Jan 31 18:34:32 CET 2012
JVD_HOME=/home/alicia/IndigoWorkspace/JavAdaptor_indip
JVD_PROJ_DIR=/home/alicia/JavAdaptor_indip/Projects
JVD_RESOURCES_DIR=/home/alicia/JavAdaptor_indip/Resources

Figure A.1: Example of a conf.init file for JavAdaptor

JVD_PROJ_DIR. The first one points to the directory containing the file that were in the metadata directory. The second variable points to the directory where metadata about projects registered in JavAdaptor are saved.

Each project is saved in a separate directory named like the project it refers to. In each project are saved those files JavAdaptor expected to find in its directory, but now there is a copy of them for each project. A new file has been introduced: project.info. Figure A shows the project info for the SnakeDemo project.

```plaintext
name=SnakeDemo
binDir=/home/alicia/PatchedWorkspace/SnakeDemo/bin
metadataDir=/home/alicia/JavAdaptor_indip/Projects/SnakeDemo
main=basics.Main
classpath=/home/alicia/PatchedWorkspace/bin
port=8080
projectStructure=/home/alicia/JavAdaptor_indip/Projects/SnakeDemo/projectStructure.xml
baseDir=/home/alicia/PatchedWorkspace/SnakeDemo
workspace=/home/alicia/PatchedWorkspace
```

Figure A.2: Example of a project.info file of the Snake Demo project

**Projects organization** JavAdaptor was not designed to handle multiple projects simultaneously, so they are treated separately from the core application. In the eye of the JavAdaptor core there is no project concept: it expects to find certain files and certain settings scattered around multiple classes. The main reason is that it expected the Eclipse .project file to be always available. In order not to heavily modify the core classes, we decided to have a single active project, whose settings are referred from the core classes and then a manager that could change the current active project with a different one. This substitution is likely not to be necessary in most cases, because a developer usually works on a single project at a time: it is more likely to think that the project chosen on the JavAdaptor startup would be the only project used for that session.
The ProjectManager class is the class which handles all the projects. This class loads the projects it finds in the JVD_PROJ_DIR directory. An existent project can be called by simply setting it as the current project with the setCurrentProject(String name) method and then it can be accessed through the getCurrentProject() method.

A new project can be created by using the method

buildNewProject(String name, String bindir, String main, int port), where:

```java
/**
 * @param name Name of the new project. Must be the name of his base directory.
 * @param bindir Path to the bin directory.
 * @param main Name of the main class.
 * @param port Port for running the application in debug mode. Default port is 8000.
 */
```

Figure A.3: Parameters of buildNewProject

The ProjectLauncher class is a class introduced to configure and launch a new application with the right settings needed by JavAdaptor, being the process not so trivial. The launcher launches the current active project. It takes care of making backup copies of the class files and an extra copy (the one with the _run suffix) where to perform the actual code injection and then starts a new process with the right settings.

As a static Reload class was already present to hide the more complicated command to start a reload, an analogous class Connect was made to shield the connect and disconnect operations.

Also an AlertBoxFactory class was built to mimic the methods offered by the Eclipse GUI with swing components and the method calls have been redirected to this class. This class can be modified to disable all output if desired.

An example of JavAdaptor main has been inserted too. It is a demo that launches the SnakeDemo project, connects to it, executes the first two reloads and then disconnects from the SnakeDemo.
Bibliography


[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edition, November 1994.


